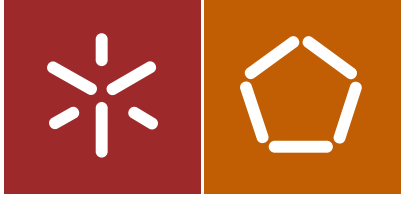
Universidade do Minho
Escola de Engenharia

Diogo Alexandre da Silva Lima

Porting Sloth System to FreeRTOS for ARM Multicore

novembro de 2015

Porting Sloth System to FreeRTOS for ARM Multicore
Diogo Alexandre da Silva Lima
UMinho | 2015



Universidade do Minho
Escola de Engenharia

Diogo Alexandre da Silva Lima

Porting Sloth System to FreeRTOS for ARM Multicore

Dissertação de Mestrado
Ciclo de Estudos Integrados Conducentes ao Grau de
Mestre em Engenharia Eletrónica Industrial e Computadores

Trabalho efectuado sob a orientação do
Professor Doutor Adriano Tavares

novembro de 2015

Declaração

Nome: Diogo Alexandre da Silva Lima
Endereço Eletrónico: diogo.11.lima@gmail.com
Telemóvel:
Bilhete de Identidade/Cartão do Cidadão: 14175186
Título da Dissertação: Porting Sloth System to FreeRTOS for ARM Multicore

Orientador: Professor Doutor Adriano Tavares

Ano de conclusão: 2015
Mestrado Integrado em Engenharia Eletrónica Industrial e Computadores

Universidade do Minho, _____/_____/_____

Assinatura: _____

Acknowledgements

It has been a long trip and many people helped me through it. Firstly my parents, Maria and António Lima, who always provided me all the support that I needed and always pushed me to get the best education possible, to ensure a better future than the one they had for themselves. A special mention to my father, who departed at the very end of this journey. I'll always look up to you and strive to be as good of a person and father as you always were.

Secondly, to my supervisor Professor Adriano Tavares, for all his guidance, trust and support put in me, towards my development as an engineer and the execution of this dissertation. A special thank you goes also to my "co-supervisor" PhD Student Sandro Pinto, for always being available to share his knowledge, support and motivation. Also for the patience shown throughout this dissertation, whenever the anxiety got the best of me.

To the Embedded Systems Research Group, from the Electronics Department from the University of Minho, that welcomed me and provided all the tools and knowledge necessary to successfully finish this dissertation.

To my fellow colleges and friends from the University, specially to Raphael Gonçalves and Carlos Fernandes, for all the time shared as friends and colleagues, bouncing ideas of each other and pushing each other to achieve our goals.

To my group of friends, António, Daniel, Fábio, João, Márcia, Mikel, Ricardo, Tiago, Vieira and Zé, for all the great times together and for supporting me, throughout the best and worst moments.

Last, but certainly not least, my girlfriend, Rita Mota, for always being there for me, through good and bad moments, and encouraging me to keep working towards my objectives.

Thank you so much, everyone!

Abstract

The microprocessor industry is in the midst of a dramatic transformation. Up until recently, to boost microprocessors' performance it was solely relied on increasing clock frequency. Nowadays, however, the power consumption requirements, coupled with the growing consumer demand, made the industry shift their focus from singlecore to multicore solutions, which offer an increase in performance, without a proportional increase in power consumption. The embedded systems field is no exception and the trend to use multicore solutions has been rising substantially in the last few years.

Managing control flow is one of the core responsibilities of an operating system. Bearing this in mind, operating systems suffer from the existence of a bifid priority space, dictated by the co-existence of synchronous threads, managed by kernel scheduler, and asynchronous interrupt handlers, scheduled by hardware. This induces a well-identified problem, termed *rate-monotonic priority inversion*. Regarding safety-critical real-time systems, where time and determinism play a critical role, the inherent possibility of delayed execution of real-time threads by hardware interrupts with semantically lower priority can have catastrophic consequences to human life.

Within this context, this dissertation presents the extension of a previous 'in-house' project, by proposing the implementation of a unified priority space approach (SLOTH) in a multicore environment. To accomplish this, it is proposed the offloading of the scheduling decisions and synchronization mechanisms to a Commercial Off-The-Shelf (COTS) hardware interrupt controller (removing the need for a software scheduler) on an ARM Cortex-A9 MPCore platform.

Keywords: Priority Space Unification, Threads as Interrupts, Multicore, FreeRTOS, ARM Cortex-A9 MPCore, GIC.

Resumo

A indústria de microprocessores está envolta numa transformação dramática. Até recentemente, para impulsionar a *performance*, a indústria dependia somente do aumento gradual da frequência de relógio. Atualmente, os requisitos de consumo energético, conjugados com as crescentes exigências do consumidor, levaram a indústria a mudar o seu foco de soluções *singlecore* para soluções *multicore*. Estas oferecem um aumento substancial de *performance*, sem o proporcional aumento de consumo energético, caracteristico das arquiteturas *singlecore*. Os sistemas embebidos não são excepção e a tendência para a utilização de soluções *multicore* tem aumentado substancialmente nos ultimos anos.

Uma das principais responsabilidades de um sistema operativo é a gestão do fluxo de controlo. Neste contexto, os sistemas operativos sofrem da existência de um espaço de prioridades bifurcado, caracterizado pela existência de tarefas, geridas pelo escalonador do *kernel* (*software)* e de interrupções, escalonadas por *hardware*. Introduz-se, assim, um problema bem identificado na comunidade científica, denominado *rate-monotonic priority inversion*. Em sistemas de tempo real, em que a segurança assume um papel fulcral e onde a *performance* e o determinismo são essenciais, a possibilidade da execução de tarefas de elevada prioridade ser atrasada, por interrupções de *hardware* com prioridade semântica inferior, pode ter consequências catastróficas para a vida humana.

Neste sentido, esta dissertação apresenta a extensão de um trabalho anterior, propondo a implementação de um espaço de prioridades unificado (Sloth), num ambiente *multicore*. Assim sendo, é proposto o *offloading* do escalonador e mecanismos de sincronização para o controlador de interrupções (*hardware)* numa plataforma ARM Cortex-A9 MPCore.

Palavras-chave: Unificação do espaço de prioridades, tarefas como interrupções, *Multicore*, FreeRTOS, ARM Cortex-A9 MPCore, GIC.

Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

This chapter contextualizes the dissertation. Section 1.1 contextualizes the problem in the embedded systems domain. Section 1.2 explains the motivation, as well as the objectives of the work. The last section (1.3) presents the overall organization of the chapter contents throughout the dissertation.

1.1 Contextualization

Embedded systems are widespread in our societies and represent a huge part of innovation in today's technology. From home appliances to factory control, automotive, medical and aerospace systems, they are present in every aspect of everyday life. Within this context, it comes as no surprise that 98% of microprocessors' annual production is used in embedded system applications [1]. The ever increasing demand of faster and better embedded systems, coupled with their mass proliferation, led to an exponential increase in embedded systems' complexity.

In the development of embedded systems there are five constraints that need to be taken into account: (i) performance, (ii) power consumption, (iii) size and weight, (iv) time to market and (v) bill of materials [2]. Despite the aforementioned constraints, embedded systems are evolving continuously, adding more and more features that up until recently were only implemented in general purpose systems.

With the challenges that the embedded system industry faces, a few technologies emerge as the most viable solutions to address them: (i) multicore, being the only viable solution for increased throughput while maintaining acceptable power

consumption, (ii) *Field-Programmable Gate Array* (FPGA) supporting hardware-software co-design and (iii) virtualization, allowing the parallel execution of multiple operating system instances on a single physical platform[3].

Everyone involved in the microprocessor industry knows Moore's Law. Not long ago, as a way to describe a microprocessors' performance, it was regularly used its clock frequency, since for many years it was the main factor to boost performance. In spite of the growing difficulty in increasing performance solely based on clock frequency, the idea of multicore systems was frowned upon, due to its inherent difficulty and complexity.

> "If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?" – Seymour Cray, *Father of Supercomputing*

The increasing consumer demand for faster execution, coupled with the fact that singlecore solutions reached their performance ceiling in relation to an acceptable power consumption, led to a paradigm shift in the microprocessor industry, from singlecore to multicore solutions. Currently, the use of multicore is the best solution to significantly increase performance, while keeping an acceptable power consumption. The embedded systems field is no exception and the trend to use multicore solutions has been rising substantially in the last few years.

1.2 Motivations and Objectives

The Embedded systems industry is in the midst of a dramatic transformation, driven by the expansion of embedded systems into new areas of application and the growing consumer demand for these systems. This consumer demand caused embedded systems to drastically increase their complexity in the last few years, driving them to close the existing gap between them and the general purpose systems. This notwithstanding, the real-time characteristics of embedded systems still differentiates them from general purpose systems.

Real-time systems are fundamental for applications that impose temporal deadlines and deterministic behaviour. Applications with real-time constraints, not only demand the satisfaction of functional requirements, but also for the externally defined temporal restrictions. Examples of such critical systems are: air-bag deployment, medical imaging systems, industrial control systems and much more. Generally, applications that require a real-time response are supported by a real-

time operating system (RTOS), that allows concurrent execution of different tasks and is designed to meet those temporal requirements. There are multiple RTOS solutions in the market, such as LynxOS[4] and FreeRTOS[5].

Traditional operating systems are characterized by an aspect that restricts optimization: the hardware abstraction layer. This abstraction layer encapsulates and hides the hardware from the rest of the operating system. It is, however, very beneficial for porting the operating system across multiple hardware platforms, since it is restricted to the re-implementation of the hardware specific layer. The FreeRTOS, for instance, has been ported to over 30 different hardware platforms, which was only possible because of its division into two architectural layers: the *hardware independent* layer, responsible for a huge amount of operating system resources and behaviour, and the *portable* layer, responsible for hardware-specific processes. In essence, operating systems make a conscious decision to sacrifice potential for hardware optimizations, in order to achieve a lower engineering effort in porting an operating system to a different hardware platform[6].

Most operating systems possess synchronous tasks, which are managed by software, and asynchronous interrupt service routines, managed by hardware. The implementation of the software unit responsible for the execution flow is one of the most critical tasks for the operating system's kernel. The division of task and interrupt priorities leads to a bifid priority space, where ISRs (Interrupt Service Routines) benefit from greater priority than any task, inducing what is known as *rate-monotonic priority inversion* [7, 8]. In real-time operating systems, where time and determinism play a critical role, the inherent possibility of a low priority ISR being able to interrupt an high priority task can have catastrophic consequences to human life. Nevertheless, this is not a new problem and it has been proposed different approaches to solve this issue. One solution to this problem is the Sloth concept[9], which proposes the management of all tasks as interrupts, offloading the scheduling decisions to a Commercial-off-the-Shelf (COTS) hardware interrupt controller. This approach not only solves the rate-monotonic priority inversion by establishing a unified priority space, but also increases system performance with the creation of a more hardware-centric operating system that exploits its inherent properties.

In this sense, this dissertation proposes to solve the aforementioned problem using functional and architectural properties of the underlying hardware. Utilizing the Sloth concept to solve the priority inversion issues, furthermore this approach

makes use of hardware COTS to boost performance while decreasing the memory footprint. Following the trend of multicore within the embedded systems field, the system will be extended to a multicore environment, to achieve even higher performance. The resulting operating system shall run on a commodity off-the-shelf microcontroller, in this case, the ARM Cortex-A9 MPCore[10].

In summary, the main objectives to accomplish throughout this dissertation are the following:

- The first objective is an in-depth study of the processor and generic interrupt controller architecture, as well as the FreeRTOS inner-workings, mainly the task management API and synchronization mechanisms.
- The second objective consists in the migration of the software scheduling decisions to the hardware interrupt controller. This comprises changes not only in the scheduler API functions, but also on the task management API.
- The third objective is the extension of the hardware-based operating system to multicore, similar to the MultiSloth concept [11], but applied to the FreeRTOS. Extending the singlecore Cortex-A9 to an hardware-based symmetric multiprocessing architecture and implementing the necessary synchronization mechanisms.
- Lastly, the fourth objective is the evaluation of the impact the hardware-based approach had on the operating system, not only if the rate-monotonic priority inversion problem was solved, but also the impact on system's performance and determinism.

1.3 Organization

In the first chapter a brief introduction is presented, where the purpose of this work is contextualized, the main objectives are listed and the dissertation structure is described.

In the second chapter, all the theoretical knowledge addressed throughout this dissertation is presented. Firstly, the main challenges and trends for the embedded systems field are described. Subsequently operating system architectures are analysed, with their inherent rate-monotonic priority inversion issues. Finally, unified priority space solutions are presented and multicore architectures are described.

In the third chapter, firstly, the ARM processor and generic interrupt controller architectures are presented. The following section describes the FreeRTOS, an introduction to its architecture, how the scheduling is performed and finalizing with synchronization mechanisms. Lastly, the development environment is presented, which is not only composed by the development platform, but also by the development toolchain used throughout the implementation.

The forth chapter describes the development of system components. Essentially, there are three main stages in the system development: (i) platform initialization; (ii) scheduler migration and task management API refactoring; (iii) implementation of the synchronization mechanisms.

In the fifth chapter the experimental results gathered from the tests performed are presented. To evaluate performance and determinism metrics, specific best and worst case scenarios are performed and compared with the native version of the FreeRTOS. In order to assess the impact of the unified priority space, behaviour tests are performed to determine if the rate-monotonic priority inversion issues were solved.

This document ends with chapter 6, which expresses the main conclusions regarding the implementation results, as well as presenting a few suggestions for future work, aimed at expanding and improving the work developed.

Chapter 2

State of the Art

This chapter describes the state of the art and presents a literature review in the embedded systems field. Section 2.1 explains the definition of an embedded system and outlines the trends and challenges in today's and the next generation of embedded devices. Section 2.2 introduces operating system architectures, focusing also on real-time operating systems. Section 2.3 discusses the rate-monotonic priority problem and describes a few existent approaches to solve it. Finally, section 2.4 explains the need for multicore in embedded systems as well as state of the art multicore architectures.

2.1 Embedded Systems

Embedded system is a broad and ill-defined term, and the rampant technological growth in the field has only made it harder to define these systems. While some consider them to be highly specialized systems developed to solve one problem in the fastest and most efficient way, others believe that anything that is outside the personal and supercomputer realm can be considered an embedded system. The growth in the field is undeniable, and nowadays they are present in very different applications, from the most common digital devices to very complex missile control systems, avionics, automotive, among many others. A few characteristics stand out amongst the more common descriptions:

- Comparing with general-purpose systems, embedded systems have more hardware and/or software constraints. Hardware limitations refer to processing

performance, power consumption, memory and so on. In terms of software limitations, this means fewer and smaller applications and operating systems are more limited or non-existent.

- Embedded systems are designed to perform dedicated functions. This definition is only partially accurate, as mobile devices are able to perform an increasing amount of functions and start being considered as embedded systems.

- Embedded systems have more security, reliability and determinism constraints when compared to general-purpose systems. On the one hand these constrains are very application specific and some systems require very high degrees of reliability, for instance, the car breaking control system. On the other hand, systems like smartphones require high degree of security from external sources trying to access private information.

During the development of this dissertation, the more evolutionary description of embedded systems will be used and as such, everything that is outside of the realm of personal and supercomputers will be treated as an embedded system.

2.1.1 Challenges and trends

Not so long ago, embedded systems were simple devices with long life-cycles. However, the industry is facing drastic transformation, as technology keeps evolving, embedded systems will continue to proliferate and to increase their complexity. The evolution of embedded systems leads to challenges in designing systems capable of higher performance, while maintaining or even lowering power consumption, size, weight, as well as the ever-increasing *time-to-market* constraints. To face all these challenges, three technologies emerge at the forefront presenting the most viable solutions:

- **Multicore**. With the aforementioned challenges and demands of embedded systems, multicore arises as the only viable solution to boost performance while maintaining acceptable power consumption. As a main focus of this dissertation, further analysis into the multicore solutions will be explored in section 2.4;

- **Field Programmable Gate Array**. Also referred as FPGA, are electrically reprogrammable silicon devices. By combining the advantages of ASIC

and processor based systems, these devices provide an high degree of flexibility, allowing lower software overhead, enabling migration of system software to dedicated hardware;

- **Virtualization**. Allows the parallel execution of multiple operating system instances on a single physical platform, as well as a better data security and isolation. The greatest advantages of virtualization are its provided isolation between operating systems and the support for heterogeneous operating systems utilizing the same platform, for instance, the co-existence of a general purpose system with a real-time operating system [3].

2.2 Operating Systems

An operating system (OS) is a software abstraction layer that hides the underlying hardware from the user. With the inherent complexity of direct hardware utilization, the operating systems act as an intermediary between the user and the computer hardware[12], managing control flows in the system on behalf of the application. There are multiple types of operating systems, as such, some operating systems are designed for convenience, known as general purpose operating systems, while others are designed to be efficient and this is the case of embedded operating systems.

2.2.1 Operating System Architecture

The core of the operating system architecture is the kernel, responsible for managing system resources. The operating system is divided into two parts: kernel space (privileged mode) and user space (unprivileged mode). The processor switches between both depending on what type of code is executing. Applications execute in user mode, while core operating system components run in kernel space. Furthermore, an application that needs to access features only present in kernel space, such as memory allocation, use specific kernel API (Application Programming Interface). User-mode applications run isolated, meaning they cannot affect data that belongs to other applications, which ensures that in case of an application crash, it is confined to the specific application while the operating system and remaining applications are unaffected. On the other hand, code that executes in kernel space shares a single virtual address space. This means that code running on

kernel space is not isolated, as such, if a problem occurs in an application running on kernel space the whole system can be compromised.

Operating systems follow essentially two different architectures: (i) monolithic and (ii) microkernel (also referred to as *µkernel*). The former executes all basic operating system services in kernel space, like the scheduler, interrupt controller, file system, etc. In this approach, numerous system calls are needed to allow applications to access all the services present in kernel space. The inclusion of all basic services in kernel space presents three big disadvantages: kernel size, extensibility as well as maintenance issues [13].

In this context, the *µkernel* was developed as a way to implement the most basic services in the kernel space, while everything else resides in user space (figure 2.1). Examples of operating systems following the aforementioned architectures are the GNU/Linux, monolithic-based and the QNX *µkernel*-based.

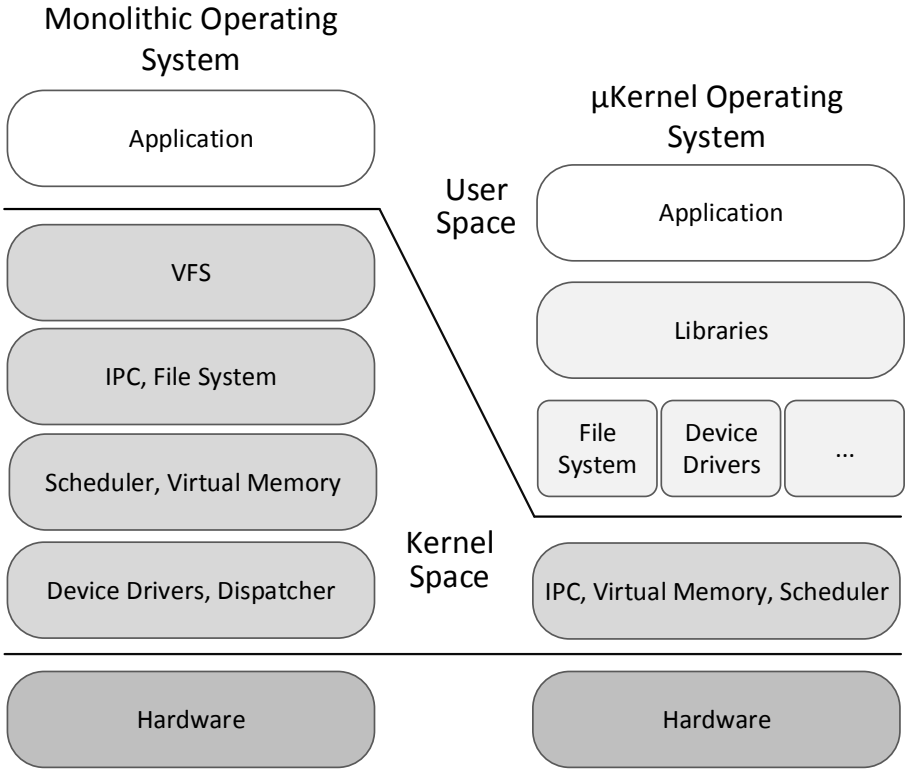


Figure 2.1: OS Architectures: Monolithic and *µkernel*

2.2.2 Real-Time Operating Systems

The word real-time in the embedded system context has a different meaning than the commonly used one. It's used to describe an application or system that has to respond to an external stimuli in a finite amount of time. Furthermore, the purpose of these systems is not throughput, but the assurance that the stipulated deadlines will be met.

The degree at which an RTOS can tolerate missing deadlines and their associated consequences defines the type of RTOS: (i) soft real-time operating systems and (ii) hard real-time operating systems. In the former a deadline occasionally not being met is acceptable and doesn't compromise the integrity of the system. However, for the latter the temporal deadline not being met is unacceptable and can cause permanent damage to the system. Figure 2.2 displays the value difference in soft and hard real-time operating systems when a deadline is not met. Given the real-time operating system prioritization of fast response times and determinism, complex scheduling algorithms are often implemented in order to achieve lower latency. Examples of such algorithms are: earliest deadline first (EDF), highest priority first (HPF) and rate-monotonic (RM).

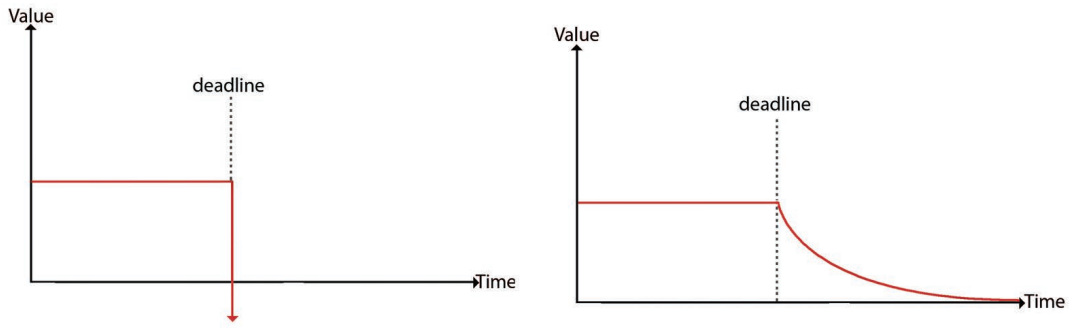


Figure 2.2: Value of real-time systems in relation to deadlines not being met

Task Management

Real-time embedded systems need to be designed with the concept of multitasking in mind. Implementing an application generally requires splitting the work among several tasks, each responsible for a portion of the application's work. This decomposition assists systems to meet performance and timing requirements. Task management involves the scheduling and context-switching of tasks within the

CPU (Central Processing Unit). In single-core solutions, this creates the illusion of having actual parallelism by maximizing the use of the CPU.

Scheduler

The scheduler is a key component of the kernel, responsible for ascertaining which task will be running next. How a scheduler decides processor time allocation has great impact on system performance, therefore scheduling algorithms have great importance. Most embedded real-time systems utilize priority-based scheduling, where each event is assigned a priority. When a new event arises, if its assigned priority is higher than the currently executing one, the currently executing task relinquishes CPU control over to the higher priority one.

An example of the control flow in preemptive scheduling is presented in figure 2.3. In the beginning, only the lower priority task is executing and no task is waiting to start executing. An interrupt request happens and the kernel saves the context of the currently running task, and services the interrupt routine. When the interrupt service routine finishes its execution, the scheduler checks for the highest priority task waiting to execute and puts it running. At the end, when the high priority task finishes, the context of the low priority task is restored and resumes execution.

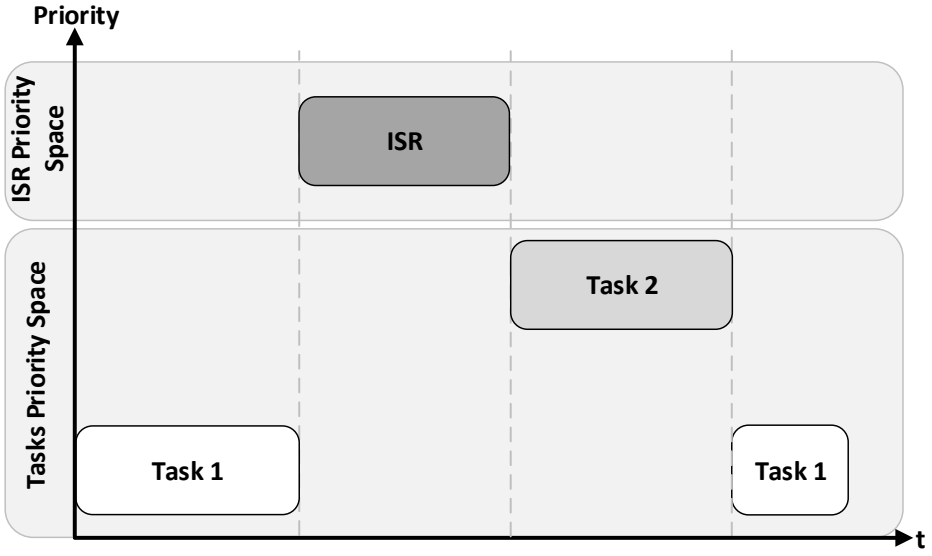


Figure 2.3: Priority-based preemptive scheduling control flow

Context-switching

Context-switching is one of the main components of any multitasking system. The context-switching basically consists of changing the executing task on the processor core. This process consists of five steps [14]:

1. Start of context-switch - Context-switching is started, either by the currently executing task yielding processor control or an interrupt handler;
2. Save Context - Processor registers are saved into the *Task Control Block* (TCB) for the task being swapped out;
3. Scheduler decision - Decision regarding which task will run next;
4. Restore Context - Processor registers are stored for the task being swapped in;
5. Resume task execution - After the task content is restored, it resumes execution.

Interrupt Management

An interrupt is an hardware mechanism provided to notify the CPU that an asynchronous event occurred. When an interrupt happens, the CPU saves the current task's context and jumps to a special subroutine, an *interrupt service routine*. The ISR is then executed and upon completion, the program returns to either the interrupted task or the highest-priority task on the queue [14].

Within most applications are *critical sections*, defined as code segments that need to have run-to-completion semantics. When a task is running a critical section, and that critical section is accessible by other tasks or ISRs, this usually warrants the disabling of IRQs to protect it. In real-time systems the disabling of IRQs should be avoided as much as possible, since this leads to an increase in interrupt latency and the possibility interrupts being missed.

Time Management

Time management is crucial for operating systems. The knowledge of the passing of time is very useful for supporting temporal services provided to the applications.

Processors provide timers that generate periodic interrupts used by the operating system the implement temporal services.

Memory Management

Regarding real-time operating systems, multiple tasks can access the same memory space. As such, the operating system needs security mechanisms to protect task code, in order to maintain memory coherency [15]. One of the key services provided is the ability to manage tasks as independent programs, running them in their own private memory space, simplifying task design of individual tasks since they don't need to know memory requirements of unrelated tasks. Furthermore, the kernel division into operating system's distinction between kernel space and user space is accomplished by using hardware components, such as the Memory Management Unit (MMU), responsible for mapping virtual and physical memory, as well as controlling memory accesses.

To sum up, memory management is responsible for the following activities: (i) mapping physical and virtual memory, (ii) allocating and deallocating memory for system tasks, (iii) dynamic memory allocation, (iv) ensure cache coherency, (v) ensure memory protection and (vi) track memory usage of system components[16].

Synchronization

The need for synchronization mechanisms, stems from the necessity for kernels to keep track of shared resources to avoid data corruption and concurrency issues. On multitasking systems, locking resources on behalf of an executing task is essential for *thread safety*, by controlling access to shared resources. Regarding single-core systems, synchronization is contained to access control within the processor core and can be as simple as *pinning* the it to a specific task. Thus, no other task will have access to the shared resource. Real-time systems accomplish this by disabling processor interrupts upon task entry in critical sections.

Concerning multicore systems, implementing synchronism mechanisms the same way as single-core solutions would not suffice. Despite access from within the processor core being restricted, it provides no protection against access from tasks running on other cores. With the preceding concerns in mind, synchronization in multicore systems needs to be refined, by utilizing synchronization mechanisms in

a way that a task will only prevent others from executing if they require access to the same shared resource, regardless of the processor core they're executing on. There are three main synchronization mechanisms:

- Built-in Atomicity. Some processor architectures provide, in their instruction set, built-in operations that are indivisible (atomic) across all cores. Therefore, it can be used to implement synchronisation mechanisms.
- Semaphore. Sometimes referred as semaphore token, is a kernel object that one or more threads can acquire or release. Kernel keeps track of the number of times the semaphore was acquired or released. If a task requests the semaphore and no semaphore tokens are available, it has to block while waiting for the semaphore to become available.
- Mutual-Exclusion (Mutex). Special kind of semaphore that allows ownership, it possesses two states, *locked* or *unlocked*. After being acquired the mutex state changes to locked and no other task can lock or unlock the mutex, until its been release by the task that currently holds control over the mutex.

Messages Passing

Operating system tasks often need to communicate with each other: *inter-task communication*. To accomplish this, tasks often use message queues. A message queue works similarly to a pipeline, temporarily holding messages from the sender until the intended receiver is ready to read them, as presented in figure 2.4. Using this method, there is a decoupling of message sending and receiving, thus allowing tasks to only receive messages when they are ready to receive them.

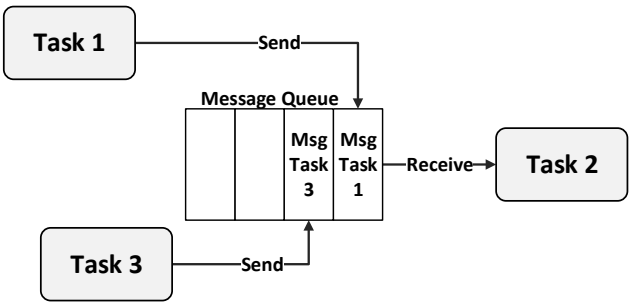


Figure 2.4: Message Queue

2.2.3 Rate-Monotonic Priority Inversion

In real-time systems, interrupts are one of the biggest causes of priority inversion. This stems from the asynchronous nature of interrupts and the fact that they can preempt any task. If interrupts preempt an high-priority scheduled event, undesired behaviour can occur [17].

Event-triggered real-time systems have a bifid priority space, where essentially two priority spaces coexist: the hardware priority space, associated with interrupt management, and the software priority space, associated with task priority management. An example of such priority space is presented in figure 2.5. Fundamentally, in real-time embedded systems featuring tasks and ISRs, high-priority software tasks can be interrupted by low-priority ISRs, inducing a well-known problem, termed rate-monotonic priority inversion. This causes loss in system scheduling precision and jitter in high-priority task execution, which can lead to missing temporal deadlines.

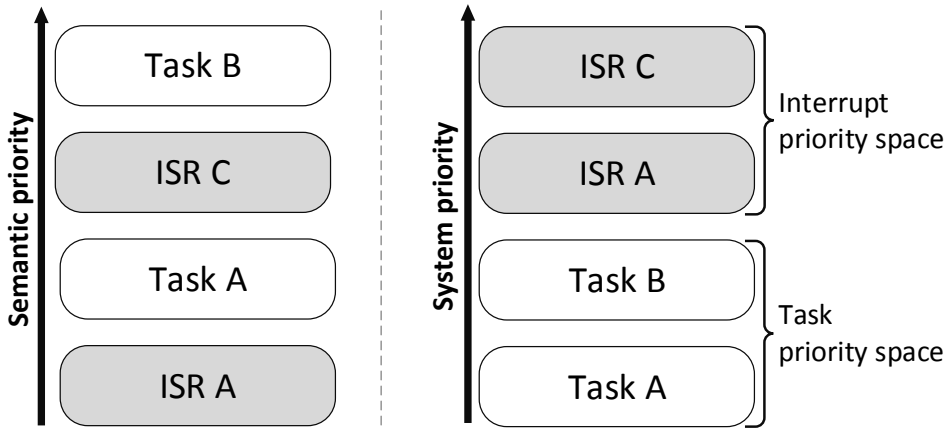


Figure 2.5: Bifid priority space example

2.3 Unified Priority Space Solutions

The previous section finished with an overview of the importance of priority and the problems of priority inversion in real-time systems. With the aforementioned priority space challenges in mind, this section presents a few solutions for priority space unification. Figure 2.6 displays the example of task priorities with an unified priority space.

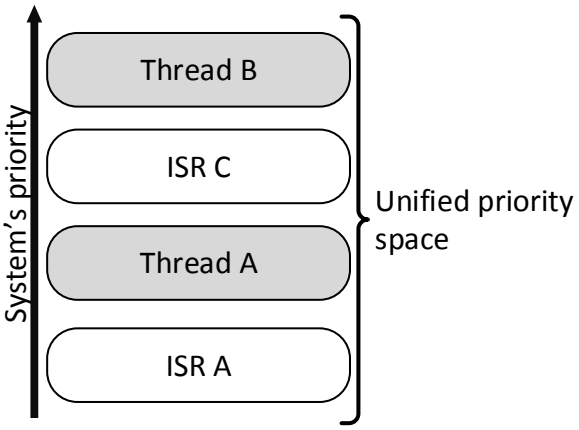


Figure 2.6: Unified priority space example

2.3.1 Interrupts as threads

Kleiman and Eykholt [18] present an approach that treats interrupts as asynchronously high-priority threads, implemented on the Solaris kernel for desktop and server systems. This concept allows interrupt handlers to use system services and introduces blocking semantics into the ISR abstraction. However, the introduced overhead to interrupt handlers makes them comparable to thread performance overheads, and still presents rate-monotonic priority inversion issues in certain scenarios.

2.3.2 Interrupt Synchronization in the CiAO

The CiAO is a project for *aspect-aware* operating systems, meaning it was developed with the concept of configurability, exploiting aspect-oriented programming. Depending on the application, the programmer can configure a variant of CiAO for his particular requirements. The general idea behind this approach is if a resource needs to be accessed by an IRQ handler that is currently being used by another IRQ handler or task, this requires the usage of *interrupt synchronization*. In this implementation interrupt service routines are split into two parts: *prologue* and *epilogue*. The former is devised for time-critical tasks that are restricted by shared resource access, and the latter can be requested by the prologue and has access to other components of the operating system, such as the scheduler. The main purpose of this division is to execute time-critical code immediately on an interrupt level and the rest later, when resources become available [19].

2.3.3 Customized hardware synthesized on FPGA or similar

Several approaches rely on *customized* hardware using FPGAs, migrating operating system functionalities to the hardware level. Although none these implementations explicitly addresses the issue of rate-monotonic priority inversion, it is prevented as a side-effect of migrating of scheduler functions to hardware. Examples of such approaches are: (i) HW-RTOS, that replaces the task synchronization and scheduler with a small hardware area[20] to increase system performance, (ii) Atalanta, implements a new multiprocessor RTOS making use of hardware/software codesign[21], (iii) Silicon TRON[22] and (iv) Task-Aware Interrupt Controller [23], which was designed with the idea of solving rate-monotonic priority inversion.

2.3.4 Sloth

Sloth introduces a more *hardware-centric* operating system by implementing all system tasks as interrupts, allowing the hardware interrupt subsystem to do most of the scheduling work. This approach allows an arbitrary distribution of priorities to tasks and interrupts, and the migration of scheduling decisions to hardware increases performance of system calls and context-switches [24]. Sloth tries to close the existing gap between operating system interface and the underlying hardware platform, adapting to the its peculiarities, instead of blindly abstracting from them [6]. Although this approach can be applicable to general-purpose operating systems, particularly suits special-purpose embedded systems, by solving rate-monotonic priority inversion issues. Furthermore, exploiting a more hardware-centric operating system is the key to achieve a higher degree of optimization.

The Sloth concept in its pure form has some limitations that have been solved with the extension of the first work:

- **Sleepy Sloth**. With the standard sloth's execution of threads as pure interrupts, threads can only use run-to-completion semantics, causing suboptimal CPU usage. This stems from the fact that, throughout its execution a thread may need to wait for externally generated signals and, instead of switching out the waiting task, the CPU has to stall until the signal arrives. Sleepy sloth presents an extension of Sloth by providing a way to overcome aforesaid limitations, implementing a new thread abstraction that combines the

advantages of the Sloth concept with a blocking functionality [25].

- **MultiSloth**. With the ongoing trend of multicore systems spreading to the embedded systems, MultiSloth extends on the original work by expanding to multicore platforms. At the same time, maintaining the crucial characteristics of Sloth, resorting to advanced synchronization mechanisms in an efficient and deterministic fashion[11].

- **Safer Sloth**. A concern arises in sloth, due to the implementation of threads as interrupts. Each thread is mapped into an interrupt handler and, since interrupt handlers run in protected space, application code is also executing on protected space. Therefore, the sloth concept was criticized for being unsafe. As one of main focus of embedded systems is memory protection, the safer sloth was designed to maintain the main characteristics of Sloth, while tackling the problem of memory protection, by offering configurable degrees of protection between tasks [26].

2.4 Multicore

The backbone of any embedded system is the processor core, responsible for instruction execution, reading encoded memory values that are mapped into instructions. Single instructions are very primitive actions, but the combination of them can create very complex system behaviour. In single-core architectures, the processor can only execute instructions *synchronously*, relying on operating system software to run multiple tasks, by means of scheduling algorithms. The speed at which processors can execute instructions and, with the implementation of the aforementioned scheduling algorithms, the impression to the user is that multiple tasks are being executed concurrently. A problem with this approach is that the more tasks trying to get processor time for execution, the less overall time each task gets. Up until the early 2000s, this problem was masked by the continued increase of processor clock frequencies, to continually squeeze performance out of singlecore processors.

With the increase in the embedded system market, demand for faster systems has risen accordingly and singlecore architectures have struggled to match the market demand. Recently, microprocessor manufacturers have realized that continuously increasing clock frequency at the expense of power consumption was not a viable

solution, as power consumption reached unacceptable levels, specially in the embedded systems field. Mobile devices have been one of the embedded systems that has grown more in the past few years. Nowadays, people often use their mobile device more than their personal computer. The increase in use of these devices, has made costumers grow to expect comparable performance and features from their mobile devices, while maintaining the expectancy for battery life the same as when mobile devices were in their early days. Multicore architectures emerge as the only viable solution to boost CPU performance with lower power consumption than their singlecore predecessors [27].

Multicore processors allow instructions to be executed *asynchronously*, each processor executes different tasks and, unlike singlecore systems, multicore systems can have *true* parallelism, thus increasing *throughput*. Another advantage of multicore systems stems from the fact that they share peripherals and power supplies across all cores, providing a less expensive solution than having multiple singlecore systems. There are mainly two types of multicore architecture: Symmetric Multiprocessing (SMP) and Asymmetric Multiprocessing (AMP). The difference between them is significant yet fairly straightforward: while in SMP systems, all CPUs are connected to a shared memory space, using the same instance of the operating system, in AMP systems each CPU executes its own instance of the operating system, and each core has its own dedicated memory area.

2.4.1 Asymmetric Multiprocessing

Asymmetric multiprocessing configuration is defined by the existence of an independent instance of an operating system in each core, as presented in figure 2.7. AMP systems can have homogeneous or heterogeneous architecture: in the former each core runs the same type and version of OS, allowing applications running on one core to communicate with applications and services from other cores, in the latter each core runes either a different OS or a different version of the same OS and communication is a lot more complex.

Typically, a process and all its threads are locked to a single processor core. This is advantageous for running legacy code, but leads to a non-optimal processor core utilization. A process and its threads can be migrated from one core to another, albeit the time costs and complexity involved makes it very prohibiting and assumes even greater complexity, if the operating systems on both ends are

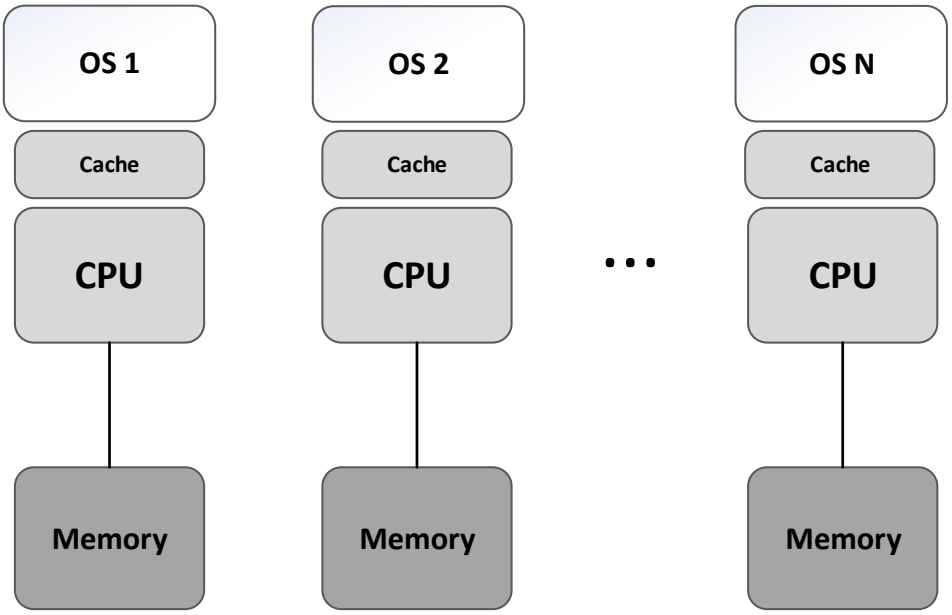


Figure 2.7: Asymmetric Multiprocessing Architecture

different.

2.4.2 Symmetric Multiprocessing

In a symmetric multiprocessing architecture all the processor cores are identical (homogeneous), employing a single operating system image, where all cores share a common memory area as well as IO devices, this is depicted in figure 2.8. In light of the symmetrical nature of system, all cores can execute code from memory, at the same time. A SMP operating system, must at all times, manage the scheduling of applications across all cores, this operation can be completely hidden from the running applications, where they have no control or knowledge of which task is running on which core at any given time. An application that is composed by several tasks that is able to run on a single core system, may very well be able to run in an SMP operating system. Nevertheless a re-structuring of the application is desirable to avoid concurrency issues and take full advantage of the SMP architecture [28]. The move to an SMP architecture does not warrant a need for the implementation of a new scheduler, what changes is the level of complexity within the scheduler. In single core architectures the scheduler has to manage several tasks within a core, on multicore it must do so across the several cores.

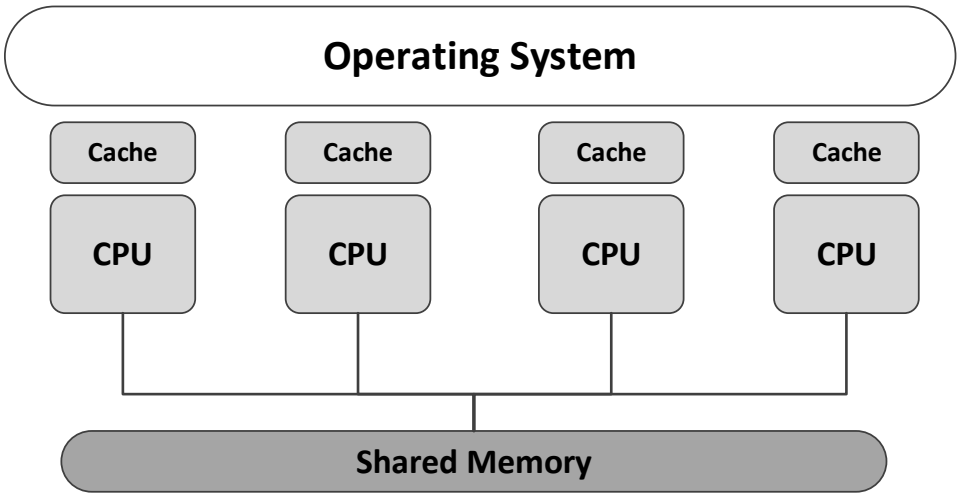


Figure 2.8: Symmetric Multiprocessing Architecture

A few concerns must be taken into account when implementing an SMP operating system. First and foremost, synchronization issues that stem from the implementation of a common memory area that is shared across all cores. Another point of concern of the SMP architecture is scalability, if too many cores attempt to access at the same point the same memory, causes a congestion in bus traffic and as such adding more cores will add little, or no performance increase to the system.

Chapter 3

System Specification

The previous chapter exposed the core concepts required for this dissertation. This chapter, in turn, presents an overview of the system, the key points of this chapter are presented, starting with the ARM architecture, its core concepts and the generic interrupt controller which is the backbone of the presented approach. After this, the FreeRTOS operating system will be analysed, presenting an architectural overview of the system, and describing how the scheduling system and the synchronization mechanisms work. To finalize the chapter an overview of the the development environment is presented: the Xilinx toolchain, ARM Fast Models and its Versatile Express model.

3.1 ARM Architecture

The key attributes of low power consumption, small size and high performance, altogether with an architectural simplicity made ARM processors very popular among embedded devices. The ARM architecture follows the *Reduced Instruction Set Computer* (RISC) architecture, incorporating its basic features as well as adding its own improvements allowing a good balance between performance, power consumption and silicon area. There have been seven major versions of the ARM architecture. The ARMv7 architecture, utilized in this dissertation, is split into three profiles: (i) ARMv7-A, (ii) ARMv7-R and (iii) ARMv7-M [29].

i. The **ARMv7-A** is application focused, running complete operating systems, characterized by high performance demand from its applications. These

processors target smartphones, tablets and infotainment systems.

ii. The **ARMv7-R** has a real-time profile, being these processors used for critical systems where reliability and predictability are decisive. These processors can be found in hard-drives, networking equipments and in critical systems like cars ABS (anti-lock braking system).

iii. The **ARMv7-M** targets microcontrollers which needs little processing power but a large amount of input and output lines and deterministic behaviour. Processors with this profile are extensively present in bluetooth devices, touchscreen controllers and remote control devices.

3.1.1 Processor Fundamentals

The ARM Cortex-A9 was designed as an high-performance and low-power processor implemented under the ARMv7-A architecture. These processors can be implemented in both uniprocessor and multiprocessor configurations, where multiprocessor configurations are provided in a cache-coherent cluster, utilizing a Snoop Control Unit (SCU). These processors provide also a set of private memory-mapped peripherals, including a global timer, watchdog and private timers for each present processor core. For the interrupt management, an integrated interrupt controller - the Generic Interrupt Controller - is available (see subsection 3.1.2).

The following features are included within the Cortex-A9 processor:

- ARM, Thumb and ThumbEE instruction set support;
- Security Extensions technology, commonly referred to as TrustZone technology;
- Advanced SIMD architecture extension to accelerate the performance of multimedia applications, for instance, 3D graphics and image processing;
- Vector Floating-Point v3, for floating-point computation (compliant with the IEEE 754 standard);
- Accelerator Coherency Port (ACP) for coherent memory transfers.

Processor Modes

The Cortex-A9 processor provides several processor modes and privilege levels defined by the ARM architecture [29], and depicted in figure 3.1:

i. **User mode** - This mode restricts use of the system resources (unprivileged execution - PL0) and so this is the mode where usually operating systems run their applications. Programs than run in user mode cannot access protected system resources, can only make unprivileged access to memory and cannot change mode, except by initiating an SVC or by external events (interrupts for instance).

ii. **System mode** - Software running in System mode (executes at PL1), has the same registers available as User mode, however having an higher level of privilege and having the possibility to access some registers not available at user mode.

iii. **Supervisor mode** - Upon a Supervisor Call (SVC) this is the default mode the exception goes to. Furthermore at the processor reset, the processor enters in supervisor mode.

iv. **Abort mode** - The default mode that a Data Abort exception or Prefetch abort takes, meaning the processor could not access the required memory location or a failed attempt to fetch a instruction.

v. **Undefined mode** - Instruction related, when an undefined instruction is taken. Usually this happens when the core is looking for instructions in the wrong place (corrupted Program Counter), or if the memory itself is corrupted. Can also occur on coprocessor faults;

vi. **FIQ mode** - Default mode taken when an FIQ occurs, in this mode registers `R8` to `R12` are banked, reducing the need to save register contents and minimizing the overhead of context switching;

vii. **IRQ mode** - Default mode taken when an IRQ occurs;

viii. **Hyp mode** - Non-secure PL2 mode, part of the Virtualization Extensions. Normally used by a hypervisor, that controls and switches between Guest OSs that execute at PL1.

ix. **Monitor mode** - Mode switched to when a Secure Monitor Call exception

is taken. This is a Secure mode, software running in this mode has access to both Secure and Non-Secure copies of system registers (only available if the implementation includes Security Extensions).

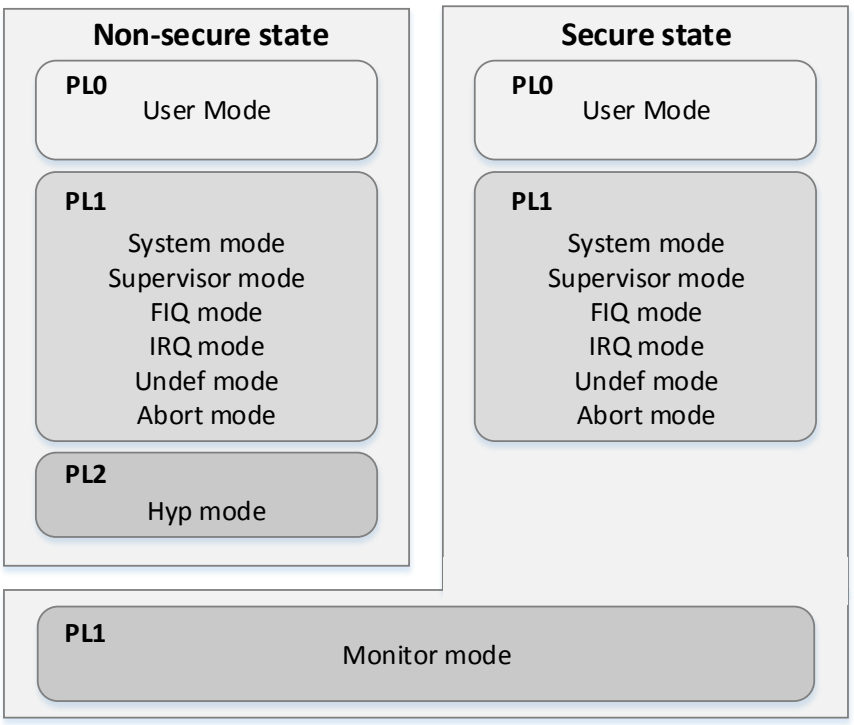


Figure 3.1: Overview of processor modes and privilege levels with security and virtualization extensions

Atomicity in the ARM Architecture

The ARM architecture provides dedicated instructions to perform atomic accesses to memory. The synchronization primitive instructions are defined as the instructions executed to ensure memory synchronization:

- `LDREX, STREX` - Load and Store Register Exclusive;
- `LDREXB, STREXB` - Load and Store Register Exclusive Byte;
- `LDREXD STREXD` - Load and Store Register Exclusive Doubleword;
- `LDREXH STREXH` - Load and Store Register Exclusive Halfword;

These instructions must be used in pairs, which means that a Load-Exclusive instructions must be used only with the corresponding Store-Exclusive.

Exceptions and Interrupts

An exception in the ARM architecture motivates the processor to suspend program execution to handle an event, entering an execution mode at PL1 or PL2, as well as the execution of a software handler for the corresponding exception. These exceptions can be: (i) reset, (ii) interrupts, (iii) memory system aborts, (iv) undefined instructions and (v) Supervisor (SVCs), Secure Monitor (SMCs) or Hypervisor calls (HVCs).

Upon the occurrence of an exception, the processor execution is forced to branch to the address which corresponds to the type of exception being handled (exception vector). The set of exception vectors for all exceptions is called exception vector table [29].

3.1.2 Generic Interrupt Controller

The previous subsection ended with an overview of the Cortex-A9 exception and interrupt handling. The present subsection presents an overview of the ARM Generic Interrupt Controller (GIC). The GIC's overview begins with an architectural overview of the interrupt controller, followed by the different types of interrupt sources and how the GIC does the interrupt handling and prioritization. Concluding the GIC overview with an explanation of the optional GIC Security Extensions and the limitations of the GIC version utilized in this dissertation, that shares the same programmers model as the PrimaCell Generic Interrupt Controller (PL390), with some implementation-specific differences [30, 31].

The GIC architecture is split into two logical blocks, a Distributor block and one or more CPU interfaces.

Distributor

The Distributor block is the first interface for every interrupt, it centralizes all interrupt sources and stores all their properties. The distributor is also responsible for dispatching the interrupt with the highest priority to the interface corre-

sponding target CPU. The programmable options provided by the CPU interface block are: (i) enabling and forwarding of interrupts to the CPU interfaces; (ii) enabling and disabling interrupts individually; (iii) setting a priority level for each interrupt; (iv) setting the interrupt target processor; (v) setting each interrupt as level-sensitive or edge-triggered; (vi) if security extensions are implemented, also provides a way to set each interrupt as Secure or Non-secure; (vii) sending software generated interrupts, SGI, to one or more target processors.

CPU Interface

The CPU Interface block provides a CPU interface for each processor, where each interface is responsible for performing priority masking and preemption handling for its connected processor. The programmable options provided by the CPU interface block are:(i) enabling the signalling of interrupts to the processor; (ii) interrupt acknowledgement; (iii) interrupt process completion indication; (iv) interrupt masking; (v) definition of preemption policies; (vi) determining the highest priority pending interrupt to be signalled to the processor.

Interrupt Sources

The GIC provides several types of interrupt sources:

- **Software Generated Interrupts**(0-15) - SGIs are interrupts generated by software, using the `GICD_SGIR` register in the Distributor and can be used for interprocessor communication. If the system implements Security Extensions interrupt 0 to interrupt 7 are used for Non-secure interrupts and interrupts 8 to 15 are used for secure interrupts.
- **Private Peripheral Interrupts** (16-31) - PPIs are peripheral interrupts that are specific to a single processor, so they are banked for every processor. These interrupts can be triggered by hardware or by writing to the `GICD_ISPENDx` register in the Distributor;
- **Shared Peripheral Interrupts** (32-1019) - SPIs are peripheral interrupts that the Distributor can route to any combination of processors. These interrupts can be triggered by hardware or by writing to the `GICD_ISPENDx` register in the Distributor;

- **Reserved** (1020-1021);
- **Interrupt Grouping** (1022) - This interrupt is only used if the GIC supports priority grouping. Indicates that there is a Group 1 interrupt of sufficient priority to be signalled to the processor that must be acknowledged;
- **Spurious Interrupt** (1023) - This interrupt is used when an interrupt that was signalled by the GIC to the processor is no longer required. When the processor acknowledges the interrupt, the GIC returns a special interrupt ID, identifying it as a spurious interrupt.

Figure 3.2 presents an overview of the GIC, showing its architectural division into Distributor and CPU Interfaces, as well as the division between types of interrupts and their connection with each associated interface.

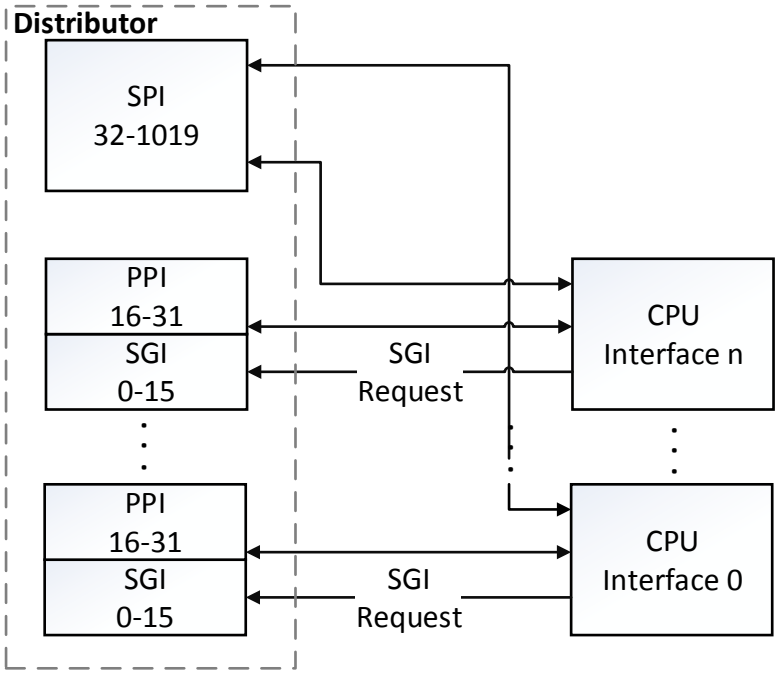


Figure 3.2: GIC Architectural Overview

Interrupt Handling and Configurations

The GIC provides four states for each interrupt: (i) inactive; (ii) pending; (iii) active; (iv) active and pending. The conditions that allow a switch in the state of each interrupt to occur are referenced in figure 3.3.

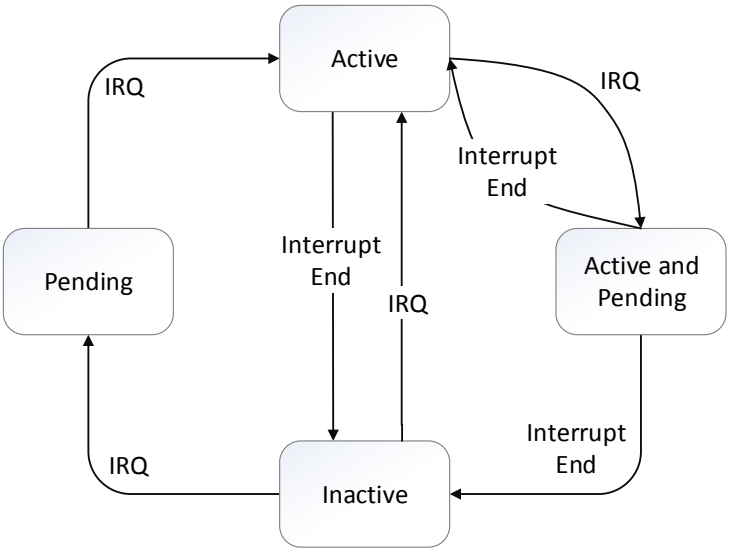


Figure 3.3: GIC Interrupt States

When an interrupt request takes place the GIC checks if the interrupt is enabled and sets it as pending. The Distributor determines the highest priority pending interrupt for every processor and forwards that interrupt for each CPU interface. The CPU interface, in turn, compares the value of the highest pending interrupt with the priority mask, and if the pending interrupt has higher priority than the priority mask value, so the CPU interface signals an interrupt exception request to the processor, as depicted in figure 3.4.

The GIC provides several programmable settings for interrupt control.

i. Allows enabling and disabling of interrupts through the `GICD_ISENABLERx` and `GICD_ICENABLERx` registers respectively;

ii. Allows setting and clearing the pending state of an interrupt using the `GICD_ISPENDRx` and `GICD_ICPENDRx` registers, respectively;

iii. Allows setting the interrupt priority using the `GICD_IPRIORITYRx` registers;

iv. Allows setting the target of each SPI using the `GICD_ITARGETSRx` registers;

v. Allows setting the interrupt as level-sensitive or edge-triggered using the `GICD_ICFGRx`

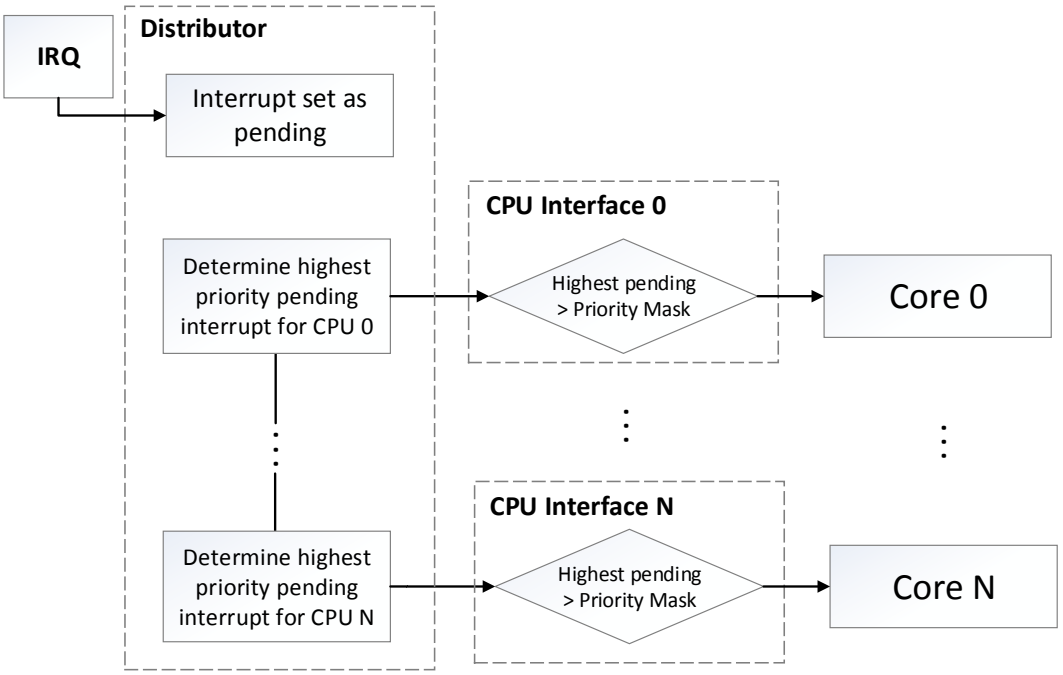


Figure 3.4: GIC Interrupt Handling

GIC Security Extensions

Security Extensions are available in the ARMv7-A architecture, and this feature facilitates the integration of hardware security mechanisms, providing Secure and Non-secure virtual memory space.

The Security Extensions implement a division of interrupts into Secure (Group 0) and Non-secure (Group 1). The behaviour of processor accesses to registers is dependent on the group associated with the interrupt. Secure accesses can read or write information regarding both Secure and Non-secure interrupts, while Non-secure accesses can only read or write information belonging to Non-secure interrupts.

Limitations

The version of the GIC present in both development platforms (see section 3.3) displays a few limitations: (i) most PPIs are reserved, only 27 through 31 are available and destined for the Global Timer, legacy nFIQ pin, Private Timers and legacy nIRQ pin respectively; (ii) the number of SPIs is limited to 64 (iii) The number of priority levels available for interrupts, the FreeRTOS provides 255

priority levels, and so the GIC should match that to provide equal number of different task priority, which is not the case since it provides less than 8 priority assignment bits.

3.2 FreeRTOS

The FreeRTOS is a real-time operating system, written mainly in the C programming language. As the name suggests the FreeRTOS is an open source, fully supported and is in constant development. First introduced by Richard Barry in 2002, it is one of the most used real-time operating systems, and has been ported to 35 architectures [5].

3.2.1 Introduction and architectural overview

The FreeRTOS presents a simple, yet very clever architectural division into two layers: hardware independent and portable (hardware dependent) layer. The former is responsible for most operating system functions, and the latter for hardware-specific processing (context-switching for instance), for this reason, the hardware independent layer remains unaltered across all ported versions. The aforementioned layered division is also apparent in the operating system's source files, as presented in figure 3.5. Not all source files are mandatory, for example, the `croutine.c`, `queue.c` and `timers.c` are optional files in the operating system, whereas all others are mandatory.

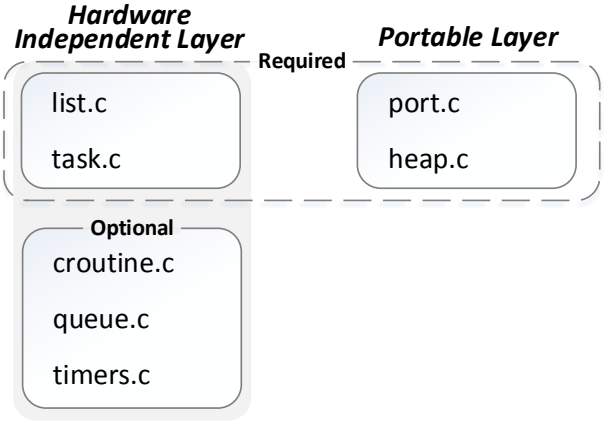


Figure 3.5: FreeRTOS architectural division

3.2.2 Tasks and Scheduling

The FreeRTOS supports fixed-priority preemptive and cooperative scheduling. In order to implement the scheduling system and related services, the FreeRTOS provides several doubly linked lists, one for each state and a few more for task management purposes (e.g., task deletion). At the core of the FreeRTOS scheduler are ready-state linked lists, where for each priority there is a corresponding linked list, that contains each task ready-to-run that has that priority. The scheduler determines the highest priority ready to be scheduled by running through the items at the head of the non-empty highest priority ready list.

Task Control Block

The Task Control Block (TCB) is the data structure that contains the task information and is used by the scheduler to identify it in queues, and retrieve and store pertinent information about the task. The FreeRTOS provides a pointer to identify the location of the TCB of the current task in execution, named *pxCurrentTCB*, as displayed in figure 3.6.

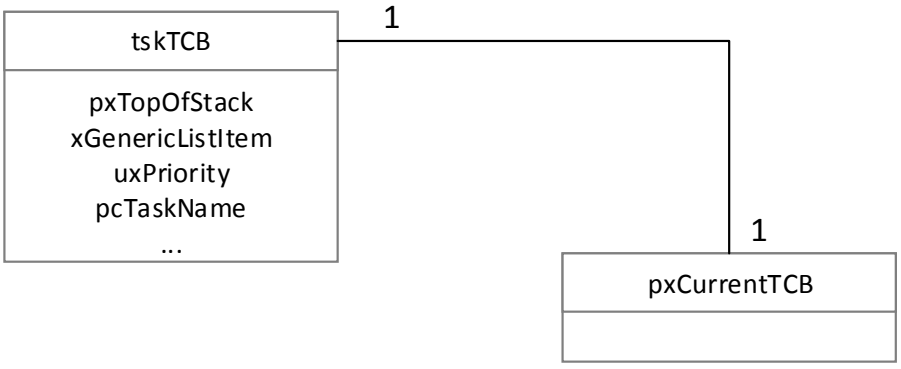


Figure 3.6: Task Control Block

Task Creation

A task is created using the `xTaskGenericCreate()` function. This entails the allocation of the task's memory, the memory belonging to the task's stack (which size is user specified in the function call) and the task control block. After the stack

is allocated, it is prepared so it can be started by the context-switching code, and at the end the task is added to the ready queue waiting to be scheduled.

Task States

Tasks running on the FreeRTOS can switch between several states, as shown in figure 3.7. The first distinction between task states is a fairly generic one, running or not-running. This distinction is not enough to fulfil all the requirements of the operating system, and so the not-running state is expanded into three different states: (i) ready, (ii) suspended and (iii) blocked.

i. *Ready* - which means a task is ready and can scheduled at any time;

ii. *Suspended* - the task was suspended by the user and is waiting for a resume to enter the ready state;

iii. *Blocked* - the task was blocked waiting on an event to enter the ready state.

iv. *Running* - state which a task takes when is currently in execution;

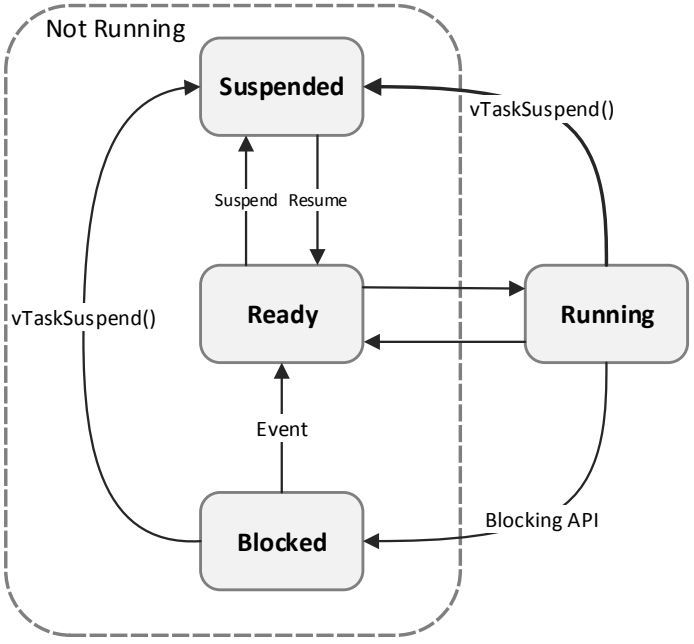


Figure 3.7: Scheduler states of the FreeRTOS

3.2.3 Synchronization Mechanisms

Synchronization is essential when developing operating systems, in order to coordinate the access to resources shared by multiple tasks. This is a fairly straightforward job on singlecore operating systems. The simplest way to avoid synchronization issues consists of: has long as the processor core is locked executing a single task, no other task will attempt to access the resource, hence no synchronization mechanism is needed since the shared resource will only be accessed by the executing task, since the synchronization is confined to the processor core. FreeRTOS accomplishes the aforementioned synchronization by disabling interrupts, preventing tasks from being scheduled on entry to a critical section. The synchronization approach mentioned is not suitable regarding multicore systems. Since the current processor core would only restrict the access to the shared resource from within the core, it would grant no protection against access from other cores.

Binary Semaphores

Binary semaphores are similar to mutexes, but have a slight difference, the fact that mutexes possess a priority inherit mechanism, this very reason makes binary mutexes mostly used for synchronization between tasks or between tasks and interrupts. These semaphores can be seen as a queue that only has two states, empty or full. Binary semaphores are created using the `vSemaphoreCreateBinary` API function, and is used with the `xSemaphoreTake` API function.

Counting Semaphores

Counting semaphores are very similar to binary semaphores, but instead of only being full or empty, they can be seen of as queues with a length greater than one. These semaphores are mostly used for counting events and resource management. To create counting semaphores the `xSemaphoreCreateCounting` is used, and the `xSemaphoreTakeFromISR` and `xSemaphoreGiveFromISR` system services are used to acquire and relinquish counting semaphores, respectively.

Mutexes

As stated previously mutexes are simply binary semaphores that include a priority inheritance mechanism, making mutexes a better choice to implement mutual exclusion. Mutexes are used as a token that guards a resource. A task that needs to access a resource must first take the mutex (acquire the token), and when it is finished using it, the mutex is given back, allowing other tasks to access this resource. If the mutex is not available when a task tries to take it, it enters a blocked state, and the time the task spends in the blocked state is implementation defined. Mutexes are created using the `xSemaphoreCreateMutex`, and share the same semaphore API functions, to allow blocking for a specified amount of time.

Priority Ceiling Protocol

Some specific cases when handling mutexes can cause priority inversion issues. For that reason, priority inherit mechanisms, are used to solve the aforementioned problem, which are displayed in figure 3.8. Whenever a high priority task attempts to take hold of a mutex that is currently held by a lower priority task (Ⓐ and Ⓑ), the priority of the task holding the mutex is risen to the point that it is re-scheduled by the operating system Ⓒ, continuing execution until it releases the mutex Ⓓ. After this, its priority is lowered to the previous priority, the high priority task waiting on the mutex is scheduled Ⓔ and acquires the mutex resuming execution [5]. This mechanism is used to not only minimize the time the high priority task is kept from executing by a lower priority task, but also to prevent deadlocks.

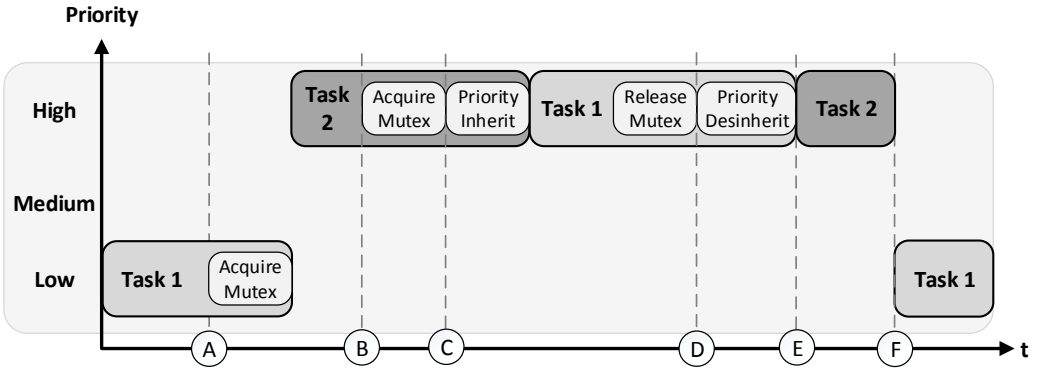


Figure 3.8: Priority Inherit Mechanism

A more complex example of the application of the priority ceiling protocol is displayed in figure 3.9. The process starts off with task C executing and acquiring the resource, then (A), task A is resumed and is scheduled, since it has higher priority than task C. Task A then tries to acquire the resource, but since it already belongs to task C, it has to block and wait for task C to give it away (B). Finally, task C continues its execution, but task B is resumed, and since it has higher priority than task C, it is scheduled (C). Which causes task A to be stuck waiting to acquire de mutex indefinitely.

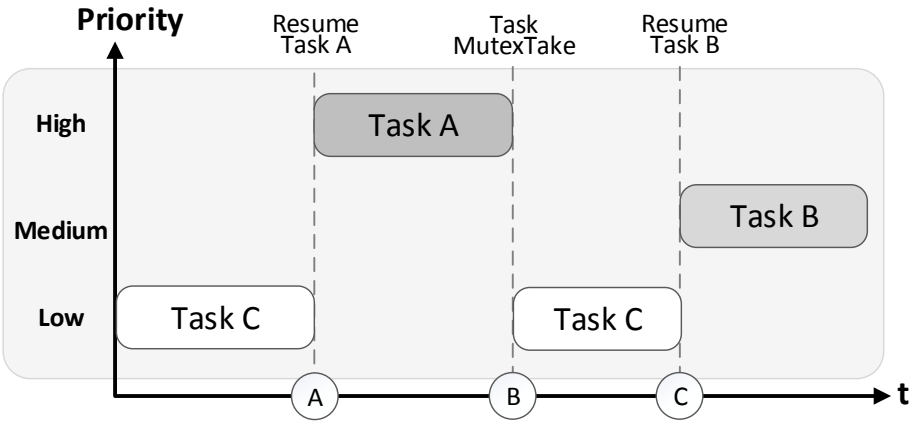


Figure 3.9: Priority inversion example in the native FreeRTOS

Figure 3.10 displays the behaviour of the FreeRTOS with priority inheritance mechanisms. The process starts off with task C acquiring the resource and continuing its execution, at (A) the task A, is resumed and starts executing. Since task A tries to acquire the resource (B), the priority of task C, which is currently holding the mutex, is raised to the priority of task A, and task A is blocked. At this point task C continues its execution until it gives the resource (C), and the resource is finally acquired by task A, and it starts executing, while the priority of task A is reset to its previous value. Followed by the scheduling of task B (D), which was resumed previously, but since it didn't have the necessary priority to interrupt task C, since its priority was raised. Finally task B ends its execution (E) and task C keeps executing, now with its original priority level.

The enforcement of the priority ceiling protocol in multiprocessor systems, contrary to uniprocessor systems, can have little to no effect in changing the event sequence or the blocking duration of high priority tasks, since not only are tasks subject to blocking from within their *binded* core, but also exposed to *remote blocking*, being blocked by a task executing in another core. These concerns regarding the

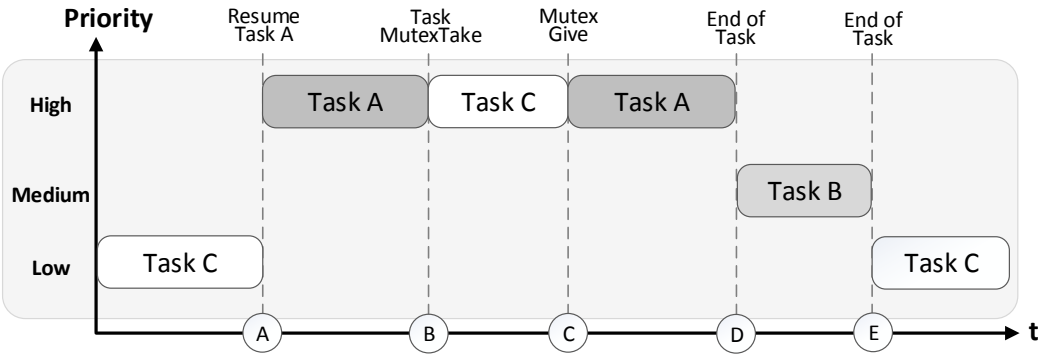


Figure 3.10: Priority inversion example in the hardware-centric FreeRTOS

use of the Priority Ceiling Protocol were presented by Rajkumar [32], as well as proposing their approach, which consists in the division of resources into local and global resources, which entails the implementation of global and local semaphores.

Recursive Mutexes

A recursive mutex can be taken repeatedly by its owned, not becoming available again until every `xSemaphoreTakeRecursive` is matched with a `xSemaphoreGiveRecursive`. For instance if a task takes a mutex 3 times, it must give the mutex back 3 times in order to make it available to be used by other tasks.

3.3 Development Environment

The following section presents the development environment, starting with the development platform, as well as a description of the toolchain used throughout the dissertation.

3.3.1 ARM Fast Models

The development of software can be delayed waiting for the development of the hardware platform. Competitiveness in today's market makes time-to-market a top concern in system development, as such, a need for a fully validated system arises, namely virtual platforms. This is the main objective of ARM Fast Models,

which allows for software development prior to hardware availability, by providing debugging, analysis and optimization of the application, consisting of high performance virtual models for ARM processors and peripherals.

The Model Debugger present in the ARM Fast Models provides high level debugging features, such as step-by-step analysis and breakpoints, as well as disassembly, register and memory views, stack, expression evaluation, while providing multicore debugging support.

3.3.2 Development Platform

During the development of this dissertation, the software deployment will be done in the ARM Fixed Virtual Platform (FVP). The deployment for debugging and data gathering using the ARM Fast Models, which provides virtual models similar to the one present in deployment platforms. Project emulation before deployment allows for a lower debugging effort and serves as a proof of concept before deploying the project in the actual hardware platform, which, given the similarities between both platforms, requires very little porting effort.

FVPs are complete simulations of ARM systems, in other words, these platforms include everything from the actual processor and memory, to the peripherals. Regarding functional behaviour, the fixed virtual platform presents the same behaviour as the actual physical platform. On another hand, the virtual model sacrifices timing accuracy, since all instructions take only one clock cycle to execute, in order to achieve fast simulation execution speed.

3.3.3 Xilinx ISE Design Suite

The Xilinx ISE Design Suite Embedded Edition development environment allows developers to boost design productivity by supporting all Xilinx programmable devices. The ISE design suite includes: (i) PlanAhead (ii) ChipScope Pro; (iii) Xilinx Platform Studio (XPS); (iv) Software Development Kit (SDK) and (v) repository of plug and play IP (intellectual property).

Figure 3.11 presents the simplified block diagram of the tools provided in the ISE Design Suite utilized during this dissertation, as well as their connections. The PlanAhead is the main tool in the embedded system development, it is responsible

to link the hardware generated by XPS to the software developed in the SDK. Both these tools are integrated in the EDK.

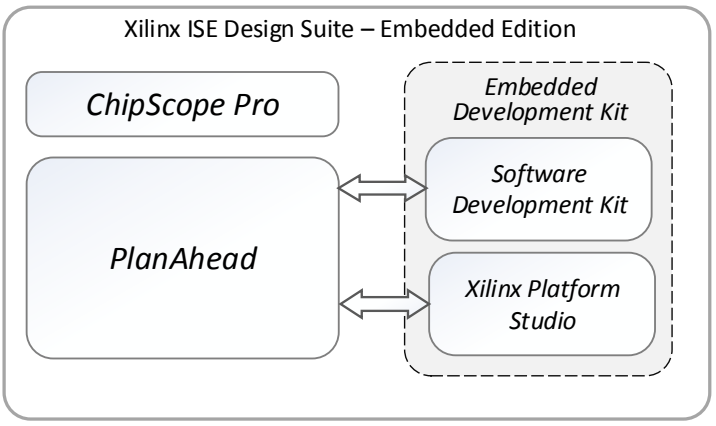


Figure 3.11: Xilinx ISE Design Suite 14.7 Embedded Edition Block Diagram

Xilinx EDK

The Xilinx Embedded Development Kit is a tool for designing embedded processing systems. Embedded systems can be very complex systems, merging the software and hardware components creates additional challenges. Xilinx EDK offers the fundamental tools and technology, coupled with a simple design flow to achieve optimal results in the development of embedded systems. Among the tools provided in the EDK are the XPS, the SDK, IP blocks and respective documentation to develop the aforementioned systems.

- ***Xilinx Platform Studio*** - Offers an integrated development environment, linking and configuration of embedded processors, from a simple state machine to a 32 bit RISC microcontroller.
- ***Software Development Kit*** - Provides a development environment for C/C++ aplications.

Chapter 4

System Development

The previous chapter presented an overview of the operating system, the ARM architecture, the processor and its inner-workings, as well as the development platform. Whereas the present chapter describes the actual system development, using the knowledge from the previous chapter towards the implementation of the different building blocks of this dissertation. Starting with the changes to the operating system initialization process, followed with the scheduler changes, the changes made to task creation and task handling, finishing with the synchronization mechanisms necessary for multicore operating systems. All this, keeping in mind, that one of the functional requirements of this dissertation is to preserve the original FreeRTOS API syntax, to allow legacy applications to be executed on this hardware-based version of the FreeRTOS.

4.1 Platform Initialization

An operating system needs a certain kind of environment before it can start executing on the underlying hardware. The following section presents this exact process, specifying the memory model adopted for the underlying platform and describing the start-up code to configure the platform for the operating system to start executing.

4.1.1 Memory Model

The first step in system implementation is the development of a linker script to provide the underlying platform's memory model for the desired operating system to run on. The memory model is an essential part of any operating system, it defines how much memory is available for the different types of data within the system. Generally, operating system address space consists of a few main segments:

- A text segment - where the executable instructions are stored in;
- A data segment - containing initialized global and static variables;
- A bss (block started by symbol) segment - which corresponds to all global and static variables that are not initialized;
- A heap segment - where the dynamic memory is allocated (using the *malloc() and free()* function, to allocate and free memory respectively);
- A stack segment - containing the locally allocated variables, function parameters and return addresses.

The memory map of the ARM fixed virtual platform is displayed in figure 4.1. Since in the virtual platform memory map the 2GB DRAM address range is *0x80000000-0xFFFFFFFF* [33], that is where the operating system memory model will be mapped.

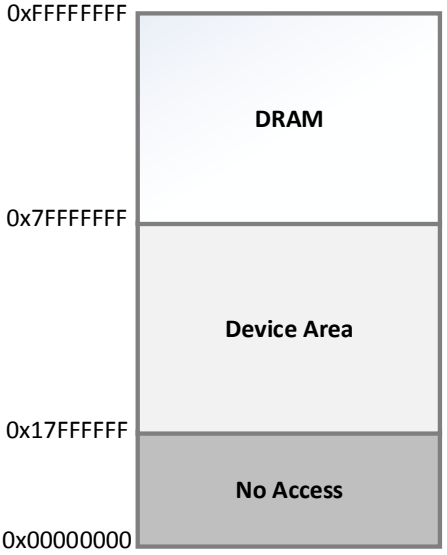


Figure 4.1: VE Memory map

The implementation of the memory model, in the linker script, begins by defining the necessary memory regions, using the `MEMORY` command, to later assign sections to particular memory regions. In this implementation, the memory sections are all defined within the `DDR` memory region, displayed in listing 1. The following steps, as displayed in the aforementioned listing, are to define the size of the heap memory region and also, define the first instruction to be executed (the entry point), using the `ENTRY` command, in this implementation it is set as the vector table.

```
1 MEMORY
2 {
3   DDR  (rwx)  : ORIGIN = 0x80000000, LENGTH =0x10000000
4 }
5
6 HEAP_SIZE = 1M;
7 ENTRY(_vector_table)
```

Listing 1: Memory regions and entry point

The next step in the implementation of the linker script, is defining the actual memory segments, using the `SECTIONS` command, as displayed in listing 2. The first segment is the `.text` section, which contains all program code, setting up the boot code at the starting address (*0x80000000*), as well as the remaining program code. The second section is the data section, which contains the initialized variables. Before placing the data section in memory, the section location needs to be specified, using the `ALIGN` command, to place the data section starting at the *0x80100000* memory address.

```
1  SECTIONS
2  {
3    startup_section : {
4      _STARTUP_START = .;
5        src/boot/boot.o (.text);
6        src/*(.text);
7      _STARTUP_END = .;
8    } > DDR
9
10   data_section : ALIGN(0x01000000){
11     _DATA_START = .;
```

```
12        src/*(.data);
13      _DATA_END = .;
14    } > DDR
```

Listing 2: Linker script - program code and data segments

The final steps in setting the memory model are: (i) placing the uninitialized variables (BSS section) immediately after the data section. The next and final step is to reserve space for the heap section in memory, which is used for dynamic memory allocation, after it is aligned to the desired starting memory address (*0x81000000*), as displayed in listing 3.

```
1  SECTIONS
2  {
3    ...
4
5    bss_section : {
6      _BSS_START = .;
7        src/*(.bss);
8        src/*(.bss.*);
9      _BSS_END = .;
10   } > DDR
11
12   heap_section : ALIGN(0x01000000){
13     _heap_start = .;
14     . += HEAP_SIZE;
15     _end = .;
16   } > DDR
17 }
```

Listing 3: Linker script - bss and heap segments

The full memory model implemented is displayed in figure 4.2. All the memory sections are presented on the left, where the entry point is set at the *0x80000000* memory address, the reset handler of the vector table is placed at that memory address. Each stack mode is presented in the aforementioned figure, where the top of stacks is set as the *0x85000000*.

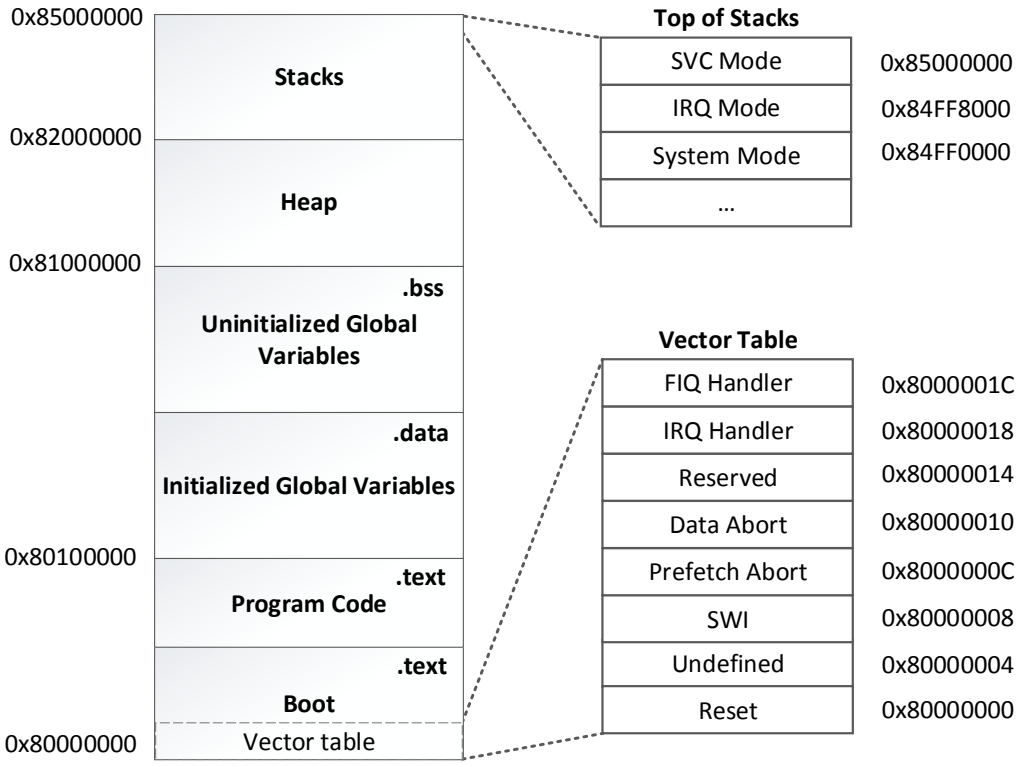


Figure 4.2: System Memory Layout

4.1.2 Start-up Code

The start-up core implements the configuration process for the operating system to take over control of the hardware platform. Regarding singlecore environments, the initialization process is very straightforward, since there is only one processor core, all the system initialization and configurations are made by that core before the operating system takes over. However, in multicore operating systems the start-up process is more complex. This increase in complexity, is due to having configurations that have to be done individually by each one of the processor cores, for instance, initializing stacks for each processor mode, while there are others that are common to all cores, so they are done only by one of processor cores, the bootstrap CPU. So, in order to implement a multicore version of the FreeRTOS changes to the initialization process were mandatory. In figure 4.3 the boot sequence implemented for the multicore version of the FreeRTOS is displayed. The boot process starts in the reset handler of the vector table, where the entry point was defined in the linker script. The first configurations are performed individually by all processor cores, setting up the stacks for each processor core.

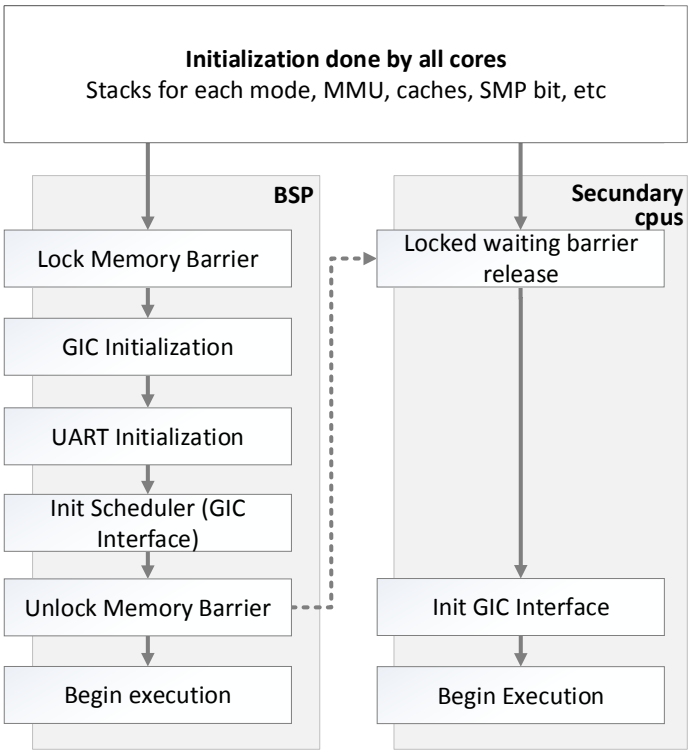


Figure 4.3: Boot Sequence for Multicore Operating System

In listing 4 an example of setting up the IRQ mode stack is presented. The `CPSR` is switched to IRQ mode and the stack pointer register (`R13`) is set by using the pre-determined size for each stack.

```
1 @@@ Set up IRQ stack pointer for each CPU
2 msr   CPSR_c,#(IRQ_MODE | IRQ_BIT | FIQ_BIT)
3 ldr     sp, =(TOP_OF_STACKS - (0x01 << STACK_BITS_PER_CPU) * (↩
     TOTAL_CPUS << 0))
4 sub     r1, sp, r0, lsl #STACK_BITS_PER_CPU
5 mov     sp, r1
```

Listing 4: Setting up IRQ mode stack

The remaining operations performed by all CPU cores are to set up the Memory Management Unit (in this case disabled), caches and the symmetric multiprocessing bit of the Auxiliary Control Register. At this point, the bootstrap processor (BSP) sets up a memory barrier (using the `DSB` and `ISB` ARM specific instructions), so that other CPUs wait until it has done all the necessary operations, secondary processors enter a two step waiting loop. Firstly, secondary cores enter a simple decrementing cycle to avoid constant memory access, which then checks

the memory position pointed by the contents of the R0 working register. The secondary CPU cores wait until the bootstrap processor writes the predetermined word, enabling them to exit the loop and branch to the operating system entry point. This branch takes the secondary CPUs to their respective idle tasks, created by the bootstrap processor.

While the secondary processors wait at the memory barrier, the bootstrap processor does all the necessary configurations to begin normal operating system functioning, which includes the initialization of the Generic Interrupt Controller and Universal asynchronous receiver/transmitter (UART). After that, the scheduler is initialized: for the purpose of this implementation the scheduler is in fact the Generic Interrupt Controller, so it is enabled in order to start scheduling tasks or interrupts. When this is finished the bootstrap processor releases the memory barrier and both begin their normal execution, enabling the interrupts in the CPSR register, and then waiting in their respective idle tasks for the GIC to schedule them tasks.

4.2 Scheduler

As stated throughout this dissertation, the concept behind this implementation is to execute tasks as hardware interrupts. Figure 4.4 presents a system overview just depicting how interrupts and tasks coexist in the same priority space: both hardware and software triggered interrupts have a configurable priority and target core. Moving all the scheduling-related decisions into the Generic Interrupt Controller, which provides each target core with the highest pending interrupt. This allows the suppression of the bifid priority space, therefore removing the problems of rate-monotonic priority inversion inherently present in the original version of the FreeRTOS. The implementation of this concept, of course, is not a simple task and requires the refactoring of many key operating system services. This section presents the implementation of the GIC's main components in order to make it work as the operating system's scheduler.

Firstly, to migrate the scheduler software functionalities into the hardware interrupt controller, the implementation of a comprehensive device driver for the GIC is imperative, not only implementing the standard hardware interrupt handling services, but also the task-associated interrupts handling services. The fundamental difference in the GIC driver implementation, involves changes to the IRQ handler.

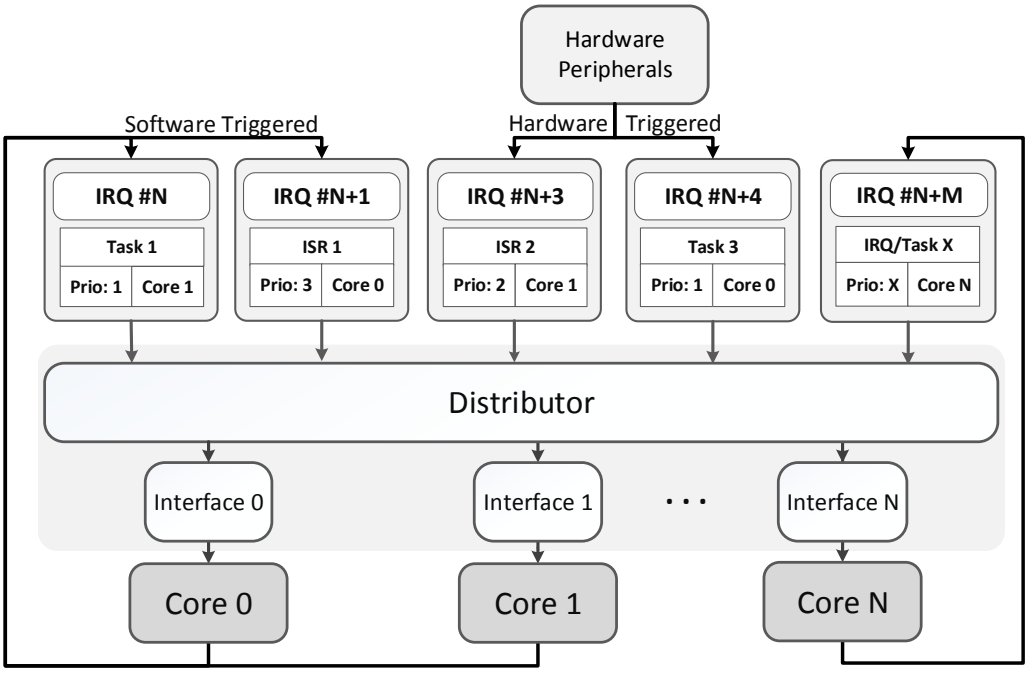


Figure 4.4: System Overview

In a standard implementation, the GIC handles interrupts in the following manner: (i) save the highest running interrupt number, (ii) use the GIC's interrupt mask to prevent lower priority interrupts from preempting the interrupt that is about to start executing, (iii) interrupt execution, and (vi) clear the interrupt and restore the previous priority mask value.

The changes to the IRQ handler, derive from the need to make the differentiation between standard hardware IRQ sources, such as timers, and task-associated IRQ sources. Since the GIC already provides reserved interrupts for the standard IRQs (see appendix B), it was only a matter of distinguishing the sources, where reserved interrupts get their task handler from the IRQ handler array, whereas task-associated IRQs call an context switch, as exemplified by listing 5. The context-switching operations not inherently done for interrupts like they were performed for tasks, also need to be implemented for context-switching between task-associated interrupts, which will be discussed later on, in section 4.3.

```
1 void irq_handler(uint32_t ack_register){
2     raw_interrupt = ack_register;
3     interrupt = raw_interrupt & INTERRUPT_MASK;
4     old_priority_mask = cpu_inter->GICC_PMR;
```

```
5     cpu_inter->GICC_PMR = (cpu_inter->GICC_RPR & PRIORITY_MASK_MASK);
6     READ_CPU_ID(cpu);
7     __asm volatile(
8         "ldr    r1, =InterruptIRQ   \n\t"
9         "mov    r3, #4              \n\t"
10        "mul    r2, r3, r0          \n\t"
11        "ldr    r2, [r1, r2]        \n\t"
12        "cpsie  i                   \n\t"
13    );
14    if(interrupt > 45)  {
15        xPortSwitchContext();
16  }
17    else{
18    if( task_irq_handler[interrupt])
19        task_irq_handler[interrupt](interrupt, source);
20    }
21    cpu_inter->GICC_EOIR = raw_interrupt;
22    cpu_inter->GICC_PMR = old_priority_mask;
23 }
```

Listing 5: GIC's IRQ Handler

4.2.1 Scheduler Start

The migration of the scheduler to hardware requires changes to the scheduler start and scheduler stop API functions. Similarly to the native version, the creation of the CPU idle function is still performed in the scheduler start API. However, since the hardware scheduler (interrupt controller) is configured in the boot process, the scheduler start simply requires the enabling of the generic interrupt controller distributor, which is shared by all CPUs, and the BSP's CPU interface, through the `GICD_CTRL` and `GICC_CTRL` registers, respectively. The secondary processors will enable their own CPU interface when the boot process finishes, and they enter their respective idle tasks. The idle tasks are also implemented as interrupts, which have the lower possible priority, so they can be preempted by any task.

4.2.2 Private GIC Functions

While the functions for standard interrupt handling remain necessary, further developments to the GIC driver were mandatory. Most private functions are necessary for creating private specialized functions for task creation and handling, as displayed in table 4.1.

Table 4.1: Private Interrupt Controller Functions

Private Functions	**Description**
`interrupt_create_task()`	Create task-associated interrupt
`interrupt_set_pending()`	Set interrupt as pending by software
`interrupt_pending_disable()`	Disable pending interrupt
`interrupt_get_free()`	Obtain free interrupt
`interrupt_priority_set()`	Count Enable Clear Register
`interrupt_target_set()`	Set interrupt target core
`interrupt_isActive()`	Check if interrupt is executing
`local_lock()`	Prevent local interrupts

This hardware-centric approach is only feasible if the underlying multicore hardware platform fulfils certain requirements. One of the core requirements is for the interrupt subsystem to support software triggering of interrupts, enabling interrupts to be synchronously activated. This is accomplished in the generic interrupt controller using the `GICD_ISPENDRx` and `GICD_ICEPENDRx` distributor registers to set and clear pending interrupts respectively, which is crucial for the implementation of the `interrupt_set_pending()` and `interrupt_pending_disable()` private functions.

4.3 Tasks

Upon implementing a fully functioning GIC driver, the next assignment is making the necessary changes to the way the FreeRTOS handles the task itself. There are two main drivers behind the changes to the system's task creation and task handling services. Firstly, to shape the operating system into using the GIC as the new scheduler, since previously it heavily relied on multiple lists to handle task states, which are no longer necessary due to the multiple states already provided by the GIC. Secondly to make the necessary changes to the operating system in order to take advantage of the multicore platform running underneath, not only

to allow tasks to run on multiple cores, but also to provide cross-core interactions between task-associated services.

4.3.1 Task Structure

There are many changes that need to take place in the task services. The first change is in the TCB structure and is the cornerstone for refactoring the FreeRTOS to work in a multicore environment. This change is fairly straightforward, and consists in extending the *pxCurrentTCB* pointer into an array of pointers to TCB, as displayed by figure 4.5, where the array size corresponds to the number processors present in the architecture. Through this modification all the advantages of the TCB structure remain unaffected, the structure remains thread-safe since each core only accesses its reserved position of the *pxCurrentTCB* array, and provides a simple solution for dealing with multiple tasks running at the same time on different CPU cores.

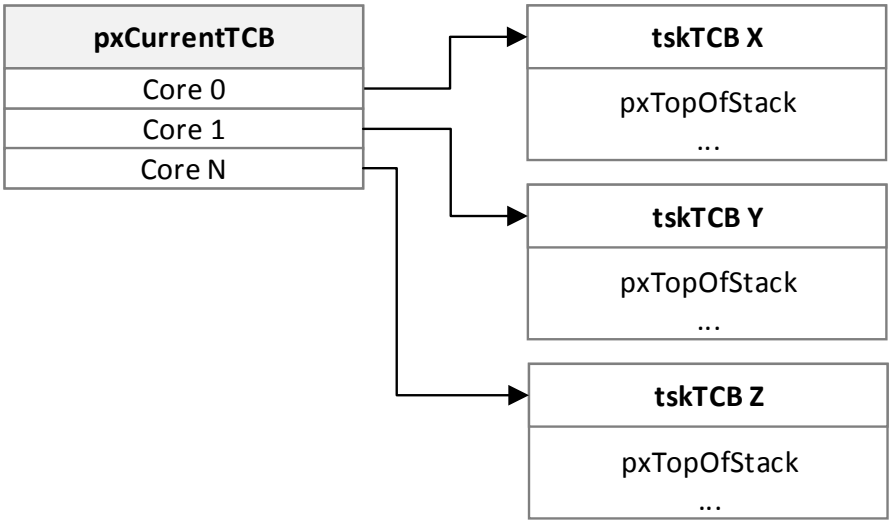


Figure 4.5: Extension to the pxCurrentTCB for Multicore FreeRTOS

The second modification to the task structure is the removal of the task-associated state lists. The original version of the FreeRTOS required several lists in order to keep track and organise existing tasks into its various states. However, since the GIC provides the necessary states for its interrupts, the task related lists become irrelevant. This modification not only speeds up task management due to the

time-intensive effort of searching through or re-organizing lists for any given task-related operation, but also lowers the memory footprint by eliminating the lists and the TCB associated fields.

4.3.2 Task Creation

The first system service that needs refactoring is the one for task creation. The original process of creating a task consists in using the `xTaskGenericCreate` API. The aforementioned routine starts by allocating the task control block and task stack, as well as initializing them. Main initializations consist in attributing the *pxCode* TCB variable with the starting function address. Furthermore, the *TopOfStack* address is calculated by using the provided stack depth and the task priority, in the *usStackDepth* and *uxPriority* variables, respectively. Following the structure presented in figure 4.6, the task function address is saved in *pxCode*, the initial Saved Program Status Register (`SPSR`) and the parameters are also saved in the task stack reserved space, finalizing by adding the newly created task to the ready list, making it immediately available to be scheduled.

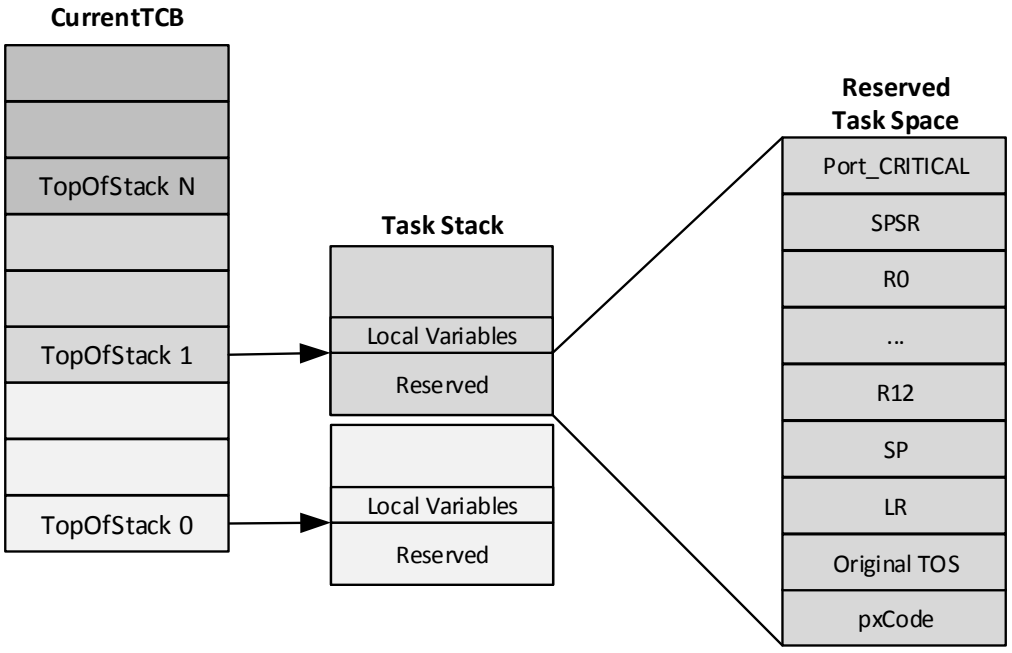


Figure 4.6: TCB and Stack structure

The modifications made to the original FreeRTOS task creation consists of, while

still performing the necessary TCB and stack allocation and initialization previously mentioned, the `xPortAssignTask()` was added to the task creation process, which is responsible for creating the task-associated interrupt, as displayed by figure 4.7. The `xPortAssignTask()` starts by acquiring an available interrupt, following a deterministic stack-like schema. This process consists of, whenever a task is created the interrupt number is popped from a stack-like structure, and whenever a task gets deleted the interrupt number is pushed onto the stack, making it available again. After an available IRQ line is found, the interrupt is configured by assigning it to the task handler, priority, target core, enabling it, and finalizing by setting the interrupt as pending, making it ready to be scheduled by its target core. If the created task has higher priority than the currently executing one, the GIC will immediately generate an IRQ so a context-switch takes place and the newly created task begins executing.

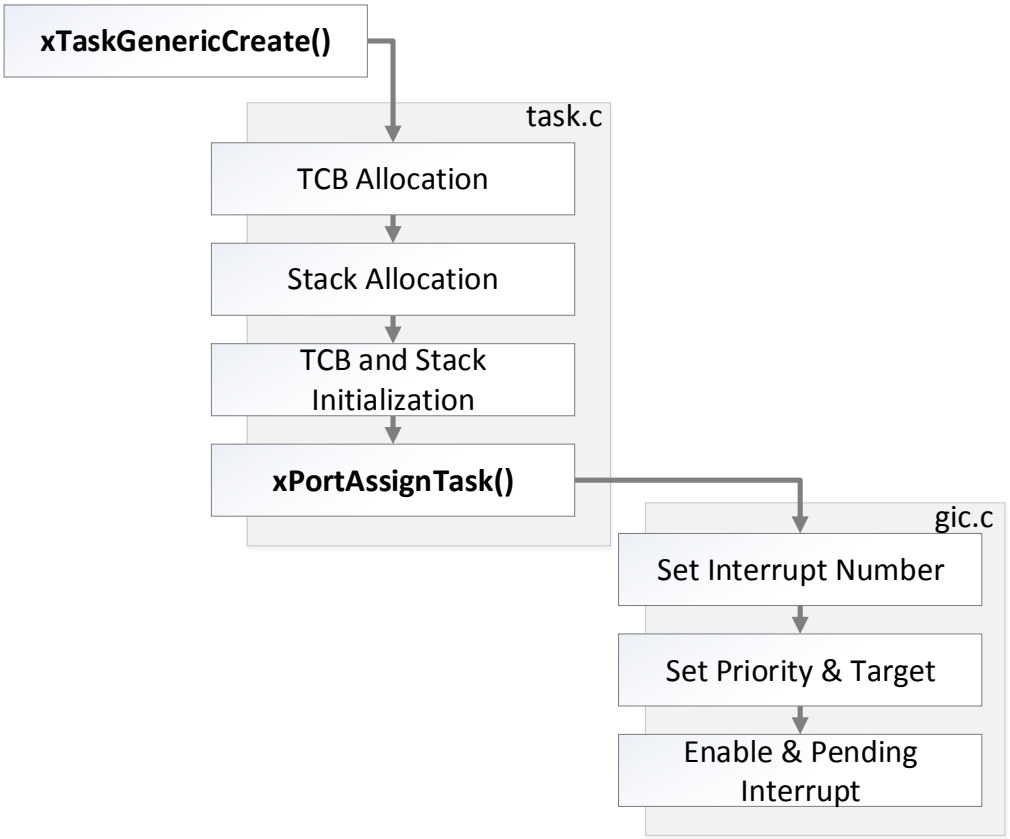


Figure 4.7: Task Creation Action Sequence

Core Assignment

Given the multicore nature of the hardware platform, it's now imperative to implement a method to assign tasks to each core as they're created, thus, a round-robin assignment schema was implemented. Upon task creation the register responsible for the target core of each interrupt, the `GICD_ITARGETSRx` register, is assigned the value of the *targetcore* variable, which is then incremented to the value of the next core. More advanced task core assignment methods and load-balancing algorithms are referenced for future work (6.2). It was also implemented an API that enables the possibility to create a task and assign it to any given core, implemented by the `xTaskCreateAffinity` which works the same way as the `xTaskCreate` function, but with the extra *uxAffinity* parameter to allow targeting the task to a specific core.

4.3.3 Task Deletion

The task deletion process works in a simple yet clever way in the FreeRTOS. When the `xTaskDelete` is called (1), the first step is to make sure the task is not currently executing, if it is, the task is forced to stop executing. After this, the task is marked as a task waiting to be deleted, and is removed from its associated list. The task allocated memory is not instantly freed, since this is a time consuming effort, which leads to indeterminism and a decrease in overall system performance. Instead, the task is placed in the task waiting deletion list, `xTasksWaitingTermination`, and when the operating system enters the idle function (2), it searches through the list of tasks waiting deletion, and if there are any listed they are then deleted, and the memory allocated to the task stack and task control block is freed. This way the API is very efficient, using the processor *free* time to perform the memory deallocation operations. The approach followed in this implementation is the same, but instead of removing the task from its list, the task-associated interrupt is cleared and disabled *on-the-fly*. This is done, so it can be allocated to another task without having to wait for the system to enter the idle function, as displayed in figure 4.8.

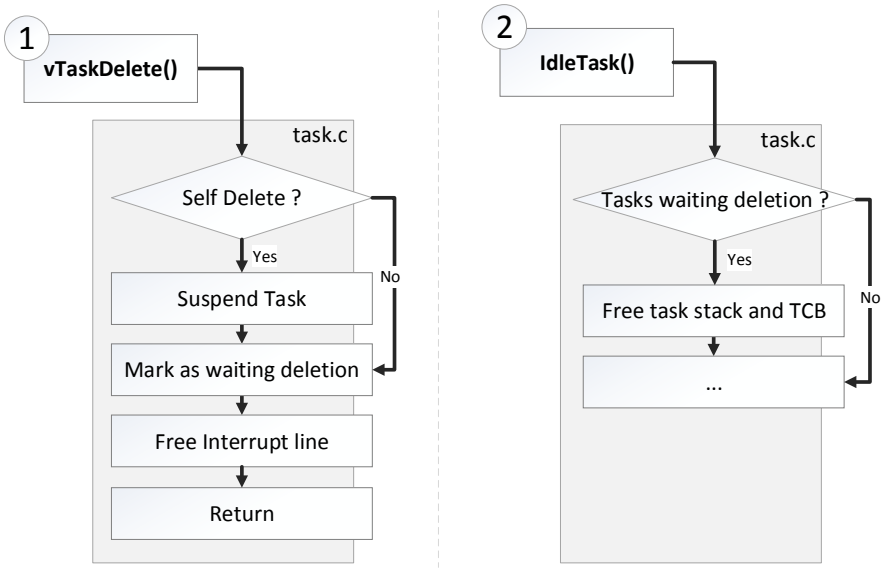


Figure 4.8: Task Delete Action Sequence

4.3.4 Task Dispatching

Task dispatching was greatly refactored in order to implement the concept used with this hardware-based approach. Since the scheduling decisions are migrated to the hardware interrupt controller and it, in turn, provides the necessary states for task management. Making use of the interrupt controller, the scheduling decisions are much faster and deterministic, but in return requires more effort in context-switching operations, considering the context save and restore is not inherently done for interrupts as it is for regular tasks on the original version of the FreeRTOS.

The actual task dispatch respects the following steps. When a new task and its associated interrupt is created, the GIC determines whether the task-associated interrupt has higher priority than the one associated with the currently executing task. If that is the case, the currently executing task is preempted by the newly created one. What this entails is a branch to the IRQ handler within the vector table, that saves the `R0` working register of both IRQ and system stack, since it is needed to perform the interrupt acknowledgement and determine its origin, as displayed in listing 6.

```
1    sub     sp,  sp, #R0_OFFSET
2    stmfd   sp,  {r0}
3    add     sp,  sp, #R0_OFFSET
4    cpsid   aif, #SYS_MODE
```

```
5     sub     sp,  sp, #0x18
6     stmfd   sp,  {r0}
7     add     sp,  sp, #0x18
8     cpsid   aif, #IRQ_MODE
```

Listing 6: Saving Offset

The next step, shown in listing 7, is to acknowledge the GIC's interrupt, by reading the `GICC_IAR` (Interrupt Acknowledge Register) from the corresponding CPU Interface and storing it in the aforementioned `R0` register. This signals the GIC the interrupt is being attended and so disables the pending state of the task-associated interrupt, in the `GICD_ICPENDRx` distributor register. The next steps adjust the link register for return correctly from the interrupt, pushes the remaining working registers and making the necessary adjustments to the stack pointer return it to the correct position.

```
1     ldr     r0,  =MPIC_BASE
2     ldr     r0,  [r0, #IRQ_ACKNOWLEDGE_]
3     sub     lr,  lr, #4
4     srsfd   #SYS_MODE!
5     cps     #SYS_MODE
6     stmfd   sp!, {r1-r3, r12}
7     sub     sp,  sp, #0x4
8     mov     r1,  sp
9     and     r1,  r1, #4
10    sub     r2,  sp, r1
11    mov     sp,  r2
12    stmfd   sp!, {r1, lr}
13    blx     irq_handler
```

Listing 7: IRQ Acknowledge and store working registers

Now a branch to the GIC's IRQ handler happens, described in section 4.2 and presented in listing 5. Assuming that the interrupt being handled is a task-associated interrupt, the `xPortContextSwitch()` routine will be called, (since the task-associated interrupt is mapped with an interrupt number reserved for task implementation), switching from the currently running task, to the one being now handled. The `xPortContextSwitch()` is responsible for all the necessary context-saving and restoring operations, not inherently done by hardware. It can be split into three main sections: the prologue, the actual body of the task and

the epilogue.

Prologue

The prologue is responsible for switching between the currently executing task to the high priority ready-to-run task that is pending, which caused the interrupt to occur. As referenced in figure 4.9, this process can be split into a few main steps. Firstly, gather in which CPU the task is currently executing (A), by reading the System Control Coprocessor's (`CP15`) Multiprocessor Affinity Register (`MPIDR`) to access the currently running task in the *pxCurrentTask* array. The second step is to perform the context-saving of the currently executing task, saving all the necessary registers in the task reserved stack space (B); Subsequently the current task ID is updated with the pending task ID, in the *pxCurrentTask* array position that corresponds to the CPU core executing the task (C); The next steps are to save the stack pointer (D) and get the address of the next task's top of stack, loading it from the *pxTopOfStack* variable to the `R12` register (E). The final steps are to load the next task's context from its task stack reserved space (F), which can be the default state attributed at task creation, or the context saved in a task suspension for instance. Lastly (G) a branch to the task *pxCode* is executed, which was popped from stack into the link register, beginning the execution of the actual task.

Epilogue

The epilogue of the context-switch routine is responsible for restoring the context, stored when the context-switch prologue took place. Figure 4.10 displays the several steps involved in a task-associated interrupt epilogue. The first step is reading the *pxCurrentTask*, to obtain the task being executed (A). The second step of the epilogue is to get the top of stack address of the next executing task, by reading the *TopOfStack* from the task *pxCurrentTCB* (B), and restoring its context, by popping all necessary registers from stack which were stored in the task's prologue (C). The following step is to get the stack pointer from SVC stack, stored during the context-switch prologue (D) and popping the *pxCurrentTask*, link register and frame pointer from task stack (E). It ends with the branch to the link register (F), which causes a branch to the IRQ handler.

In the IRQ handler function the end of interrupt is signalled by writing the inter-

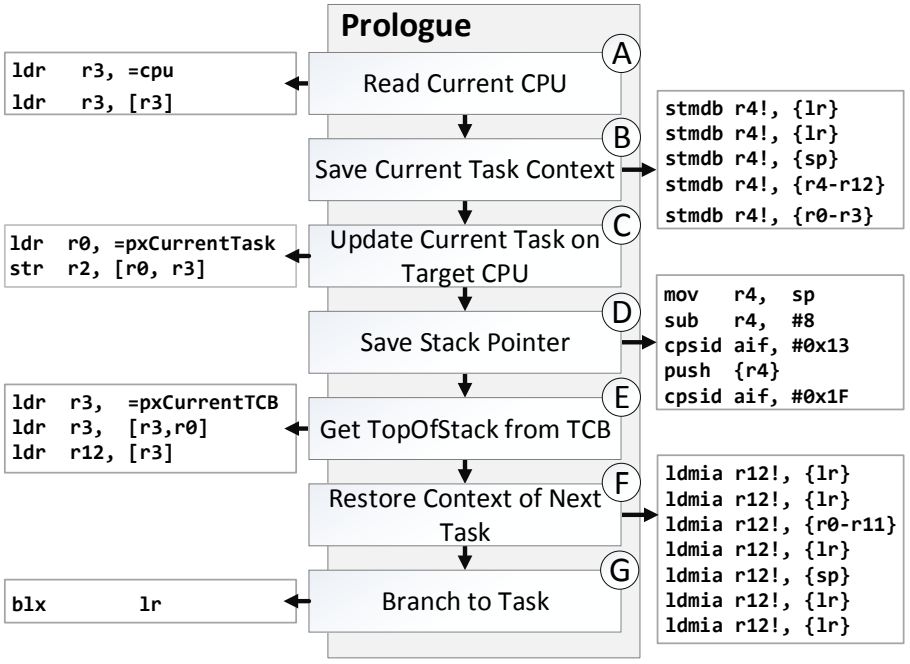


Figure 4.9: Task Context-Switch Prologue

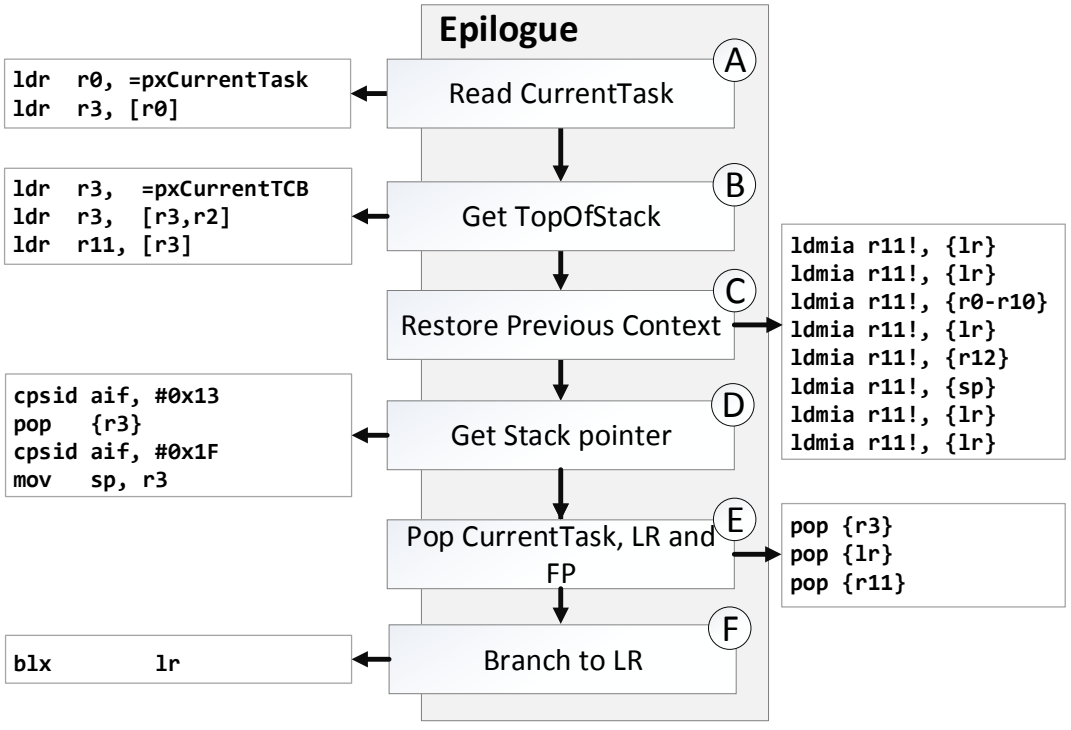


Figure 4.10: Task Context-Switch Epilogue

rupt to the GICC_EOIR, and restoring the previous priority mask by writing to the GICC_PMR, (lines 20 and 21 of listing 5). The final steps of the epilogue are done in the raw IRQ handler. The link register and stack pointer adjustment are loaded from stack, the stack is adjusted and the working registers are popped from stack. Lastly a return to the previously executing instruction before the interrupt took place is performed, as shown in listing 8.

```
1 ldmfd   sp!, {r1, lr}
2 add     sp, sp, r1
3 ldmfd   sp!, {r0-r3, r12}
4 rfefd   sp!
```

Listing 8: End of interrupt and restoring previous execution point

4.3.5 Task Priority Handling

The FreeRTOS provides the possibility of changing task priority *on-the-fly*, using the `vTaskPrioritySet()` API function. In this implementation this API function needs to be refactored, given that the GIC takes over the software scheduler's responsibilities. The implementation is greatly simplified since, while in the native version, the system was responsible for reordering the task-associated list and checking if the change in priority triggers a context-switch, in this implementation the priority change consists in changing the priority in the GIC's associated register, the `GICD_IPRIORITYRx`. The hardware interrupt controller will then take care of checking if the change in priority triggers an interrupt.

4.3.6 Task Blocking

The main drawback of executing tasks as pure interrupts, stems from the run-to-completion nature of hardware interrupt handlers. However, the possibility of suspending tasks is an essential feature of tasks within the FreeRTOS and many other operating systems. To overcome this limitation, a blocking feature was devised, by extending interrupts to behave also as threads: threads as interrupts as threads. The blocking feature is implemented by taking advantage of task specific stacks. Since multiple tasks cannot save their context on a common stack when being blocked, task specific stacks are used with a pre-determined stack reserved space for context-switching operations (presented in figure 4.6). The blocking

feature uses the original FreeRTOS API (`vTaskSuspend`), the task suspend works either a task is currently executing or if it's just waiting to be executed. For the resume, the original API (`vTaskResume`) is also used, to resume any task that was previously suspended.

Suspend

The first operation of the suspending routine is to check if the task being suspended is currently being executed. If it's not, the only step needed is to disable the pending state of the task-associated interrupt, ensuring the task won't be scheduled. On the other hand if the task being suspended is executing on the processor core that originated the suspend call, a context-switch needs to occur, saving the context of the currently executing task and restoring the previous processor context. Suspending a task was designed with a structural division into two main steps, the prologue and epilogue.

Prologue - The prologue consists of firstly changing the task state to suspended, in the task control block variable *pcTaskState*, as displayed in listing 9, the purpose of the state change is for later be used in the resume API function, to check if the task is suspended and to avoid multiple calls to suspend the same task.

```
1  vTaskSuspend()
2      ...
3      ldr  r0, =uxTaskNumber
4      ldr  r2, [r0]
5      /* Get position of the TCB within the array */
6      mov  r3, #4
7      mul  r0, r2,r3
8      /*Change task state to suspended*/
9      ldr  r3, =pxCurrentTCB
10     ldr  r1, [r3, r0]
11     mov  r3, #'S'
12     mov  r0, #4
13     str  r3, [r1, r0]
```

Listing 9: Change Task State to Suspended

The main step is saving the context of the task being suspended in the task reserved stack, similarly to the way it is implemented in the context-switching function

(listing 10).

```
1 vTaskSuspend()
2     ...
3     add   r12,  r12,#0x48
4     stmdb r12!, {lr}
5     sub   r12,  #4
6     stmdb r12!, {sp}
7     sub   r12,  #4
8     stmdb r12!, {r0-r11}
```

Listing 10: Save suspended task context

Epilogue - The epilogue consists of restoring the context to the state before the start of the task being suspended. This is fairly straightforward given the way this implementation was devised, since when the interrupt happens, the previous state is saved in task stack reserved space. This means the GIC will then take care of either scheduling a new interrupt, or the processor will continue to execute as it was previously to the suspended interrupt, as displayed in figure 4.11.

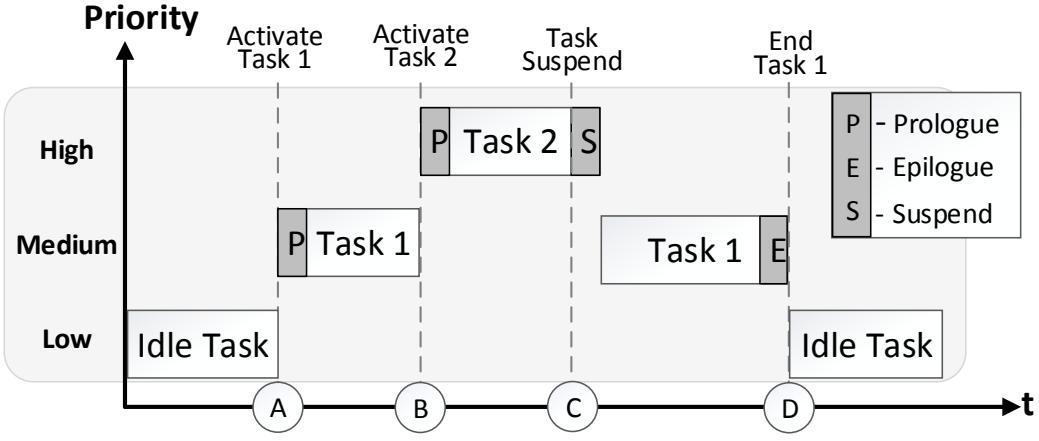


Figure 4.11: Example of the suspend control flow

Resume

Given the way the blocking feature was designed, the resume routine is quite simple, since the suspend consists in saving context and disabling the interrupt, the resume can be easily done by re-enabling the interrupt. First action is to check

if the task being resumed is set as suspended. If it is, the task state is changed to ready and secondly all that is needed is to set the task-associated interrupt as pending. When the task-associated interrupt is the highest pending, a context-switch will occur and restore the context stored when the task was suspended, as would happen with a newly created interrupt. The only difference between these two cases is, the stack reserved space is filled with the task state at the time it was suspended, continuing from the instruction it was being executed. Following the control flow example provided for the suspend function, a resume to the previously suspended task was added, as displayed figure 4.12.

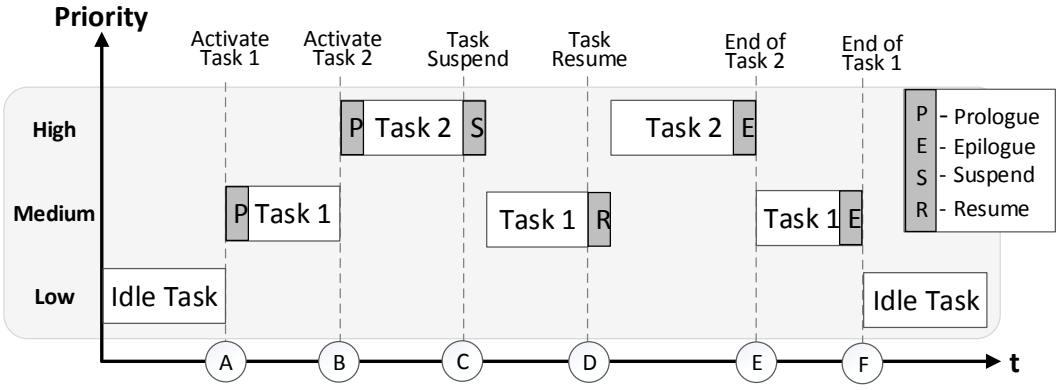


Figure 4.12: Task Resume control flow example

4.3.7 Cross-Core Interactions

To achieve optimal use of the multicore architecture, and given the way operating system services have the possibility to be called from one processor core to act on another, this is accomplished by using cross-core communication mechanisms. In this implementation, cross-core communication works by utilizing a structure, that basically works as message passing between cores: For instance, suspending a task that is running on core 1, by calling a `vTaskSuspend()` from core 0.

For this to happen, when core 0 calls the `vTaskSuspend()`, the first action is to check if the task being suspended is running on the core executing the suspend, or any other core. Subsequently if the task is running on another core, the communication structure is filled with the message needed by calling the `xWriteMessage()`. The interrupt ID, the handler of the task being handled and the operation code relating to the task that needs to be done by another core is written to the communication structure, as displayed in figure 4.13. When the communication structure

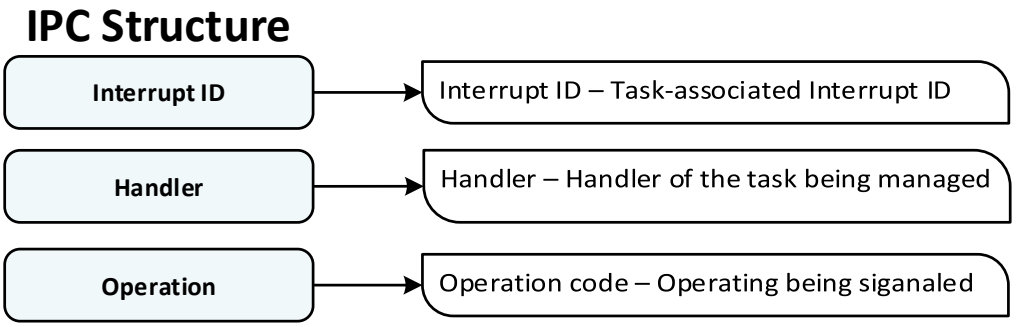


Figure 4.13: Cross-Core Communication Structure

is filled, a software generated interrupt is triggered using the `GICD_SGIR` register. The communication SGI is configured at operating system start, with the maximum priority, so it cannot be interrupted, as shown in figure 4.14.

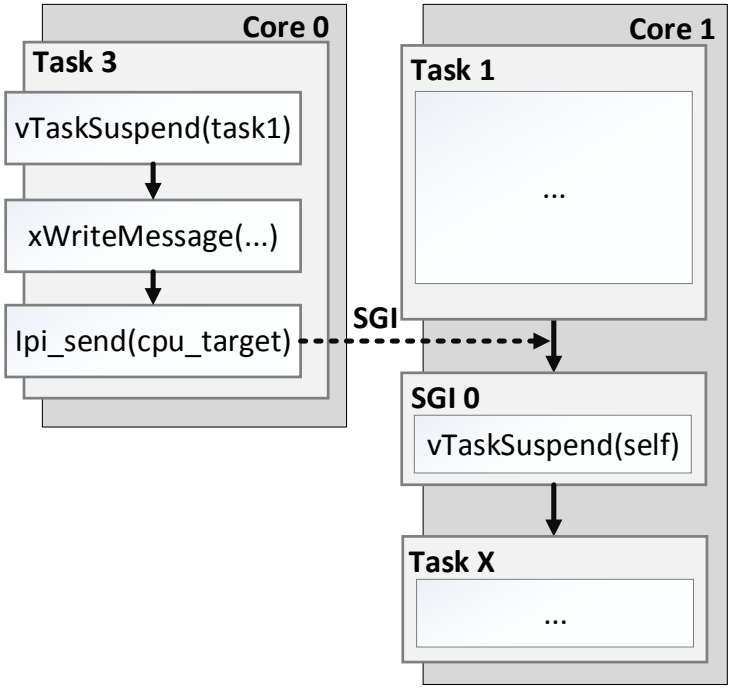


Figure 4.14: Cross-Core Interaction Example

4.4 Synchronization Mechanisms

In order to achieve predictable resource access, synchronization protocols needed to be implemented given the multicore refactoring of the FreeRTOS. Resource

sharing in real-time systems is a potential source of priority inversion problems, where low priority tasks can cause delays in high priority tasks, or even deadlocks. Uncontrolled resource access can cause high priority tasks to miss their deadlines, or even fail them. As such, real-time operating systems must implement synchronization mechanisms to ensure a controlled environment for resource sharing and limit the effects of priority inversion. This becomes specially relevant in multicore operating systems where parallel access to an indivisible resource is possible. The objective is to allow mutually exclusive access and ensuring deadlocks will not occur, while having a concern with maximizing processor utilization, by making sure tasks block while waiting for a resource to become available.

4.4.1 Synchronization Queue structure

In order to diminish the effects in determinism, stemming from the need of synchronization mechanisms in the multicore environment, the way tasks are placed in the waiting queue to access a resource was changed. Since each task has a unique priority, instead of using a list where each node corresponds to a task waiting to obtain access to the required resource, the implementation of a bitmap structure provides a deterministic management of the queue. The enqueue process simply requires a logic OR operation that enables the bit in the bitmap, corresponding to the priority of the task being set in the resource management queue. Figure 4.15 displays the way the bitmap works to enqueue a task: in this case, two tasks are trying to access the resource, one with priority 2, and another with priority 4.

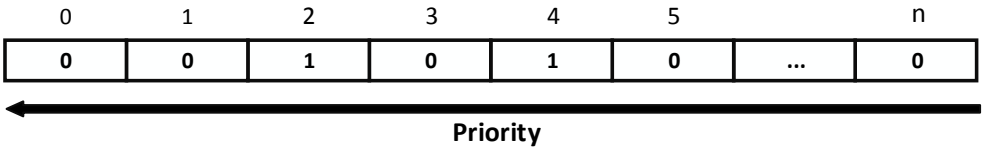


Figure 4.15: Bitmap Queue structure

The dequeuing process is slightly more complex. Although removing the task from the queue is as simple as disabling the bit corresponding to the task's priority level, it becomes necessary to obtain the next task to access the resource. The bitmap structure uses an one-hot encoding where each position corresponds to a task priority, as exemplified in table 4.2.

Table 4.2: One-hot encoding conversion table

Integer	Binary	One-Hot Encoding
0	000	000000
1	001	000010
2	010	000100
3	011	001000
4	100	010000
5	101	100000

In order to convert from one-hot encoding to obtain the task with higher priority waiting to access the resource, an assembler subroutine was created. Converting from one-hot encoding to the corresponding grant number, which means, finding the first enabled bit from the left, and then converting all the remaining bits at the right to zero, can be done by using the `CLZ` (count leading zeros) instruction. The overall process of finding the next task to access the resource is presented in figure 4.16, where tasks with priority 3 and 5 are waiting in the resource queue. The grant conversion determines the task with higher priority to access the resource is task 1 (priority 3).

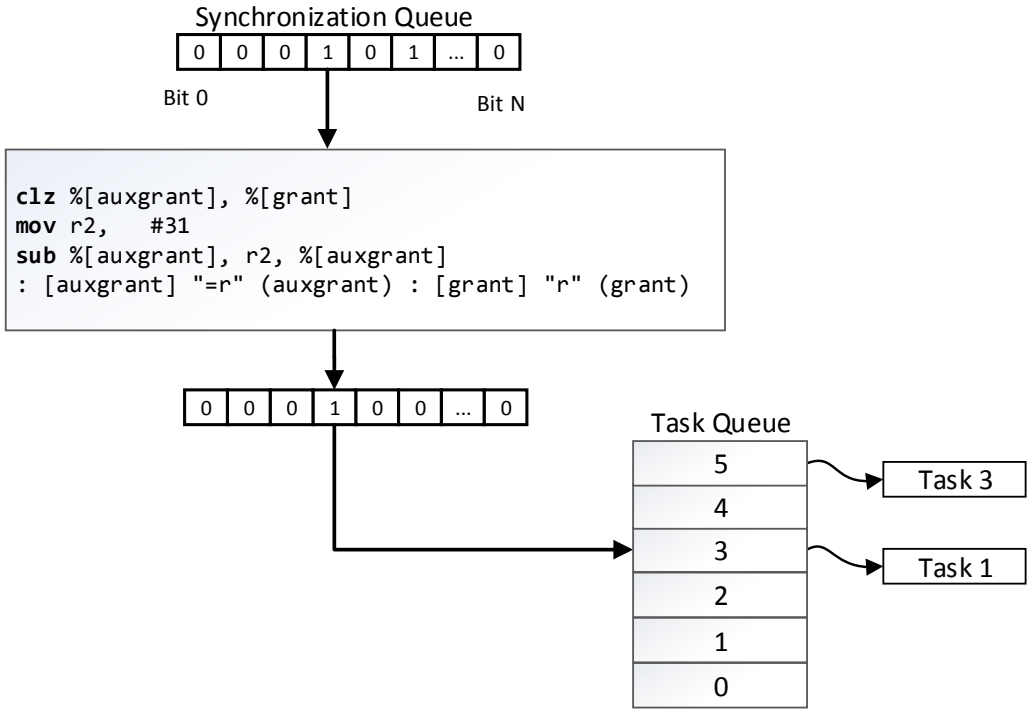


Figure 4.16: Overall Enqueue and Dequeue process

4.4.2 Local Synchronization Mechanisms

Regarding intracore resource sharing, where the resource is only accessed by tasks within a single core, the priority ceiling protocol (PCP) solves the aforementioned problem. When a task acquires a resource, the task's priority is raised to the highest priority of all tasks waiting to access the resource, preventing all other tasks that are waiting to access the resource to preempt the task that holds the resource. When the resource is released the task's priority is lowered to its previous priority, allowing it to be preempted by other tasks if their priority level is higher. This protocol works in singlecore operating systems by preventing the dispatch of other tasks that want to access the resource. To implement the local synchronization mechanisms, two new API functions are necessary, the `xAquireLocalResource()` and `xReleaseLocalResource()`, which are mapped into the `xSemaphoreTake()` and `xSemaphoreGive()` to allow portability of legacy applications.

Compared to the original version of the FreeRTOS the creation of a shared resource starts the same, by using the provided `xSemaphoreCreateMutex()` API to create the mutex. When a task requires a shared resource, the API is called. As displayed in figure 4.17, it starts off by reading the task's priority and preventing local interrupts from happening by setting the task's priority to the maximum possible priority (A). After this, the resource priority is compared to the task trying to acquire the resource, if the resource is not taken it has the lowest possible priority. If the task trying to acquire the resource has higher priority than any task currently holding the mutex, the resource's priority is raised to match the task (B). The task is then enqueued in the aforementioned bitmap (C) (presented in subsection 4.4.1). If the resource is taken by another task, the one trying to acquire it is blocked (D) waiting for the resource holder to relinquish its control over the resource, so it can be enabled. However, if the resource is not taken, it's state is set as taken, and the local interrupts are reenabled (E).

The process of releasing a resource starts off by preventing local interrupts from happening, in the same fashion as the acquiring of the resource (F). The second step is to dequeue the task from the bitmap by disabling its corresponding task priority bit (G). The next step is to obtain the *grant* value from the bitmap to determine the next task to acquire the resource (H). If there is a task waiting to acquire the resource it is then unblocked (I) and finally the interrupts are enabled again (J).

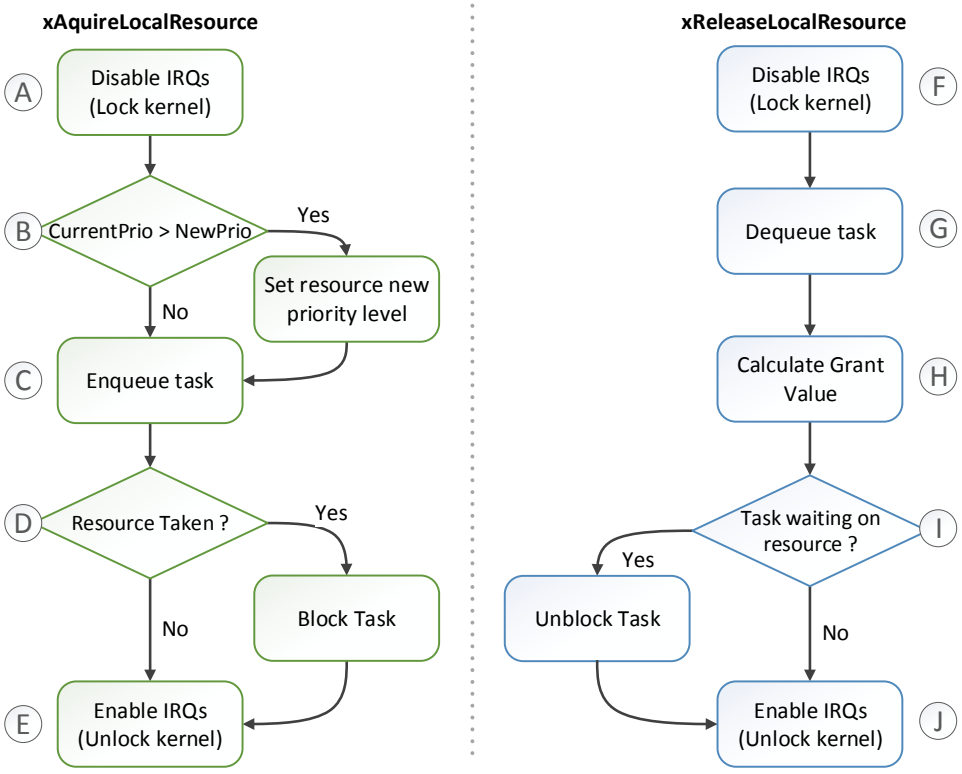


Figure 4.17: Local Resource Acquire and Release

4.4.3 Global Synchronization Mechanisms

In a multicore environment, resources can be shared across multiple cores at the same time. Therefore a global synchronization mechanism to avoid data corruption that arises from cross-core concurrent access to a resource is necessary. A priority queue that provides synchronized operations, guaranteeing no concurrent queue manipulations are performed, is the key to a multicore synchronization mechanism. To achieve atomicity in queue operations the ARM hardware exclusive access instructions (see subsection 4.4.4) are used to perform the enqueue, dequeue and observing queue data. Figure 4.18 displays the process of acquiring and releasing of a resource.

Regarding the acquisition of a resource, the `xAquireGlobalResource()` API is used. At first, the system service must be non-preemptable, so local interrupts must be disabled on the executing core (A). Before the task is enqueued the resource priority must be raised to the ceiling priority, which means the highest priority of any task trying to access the resource (B). The next steps are the essential difference from the local and global synchronization: the atomic enqueue

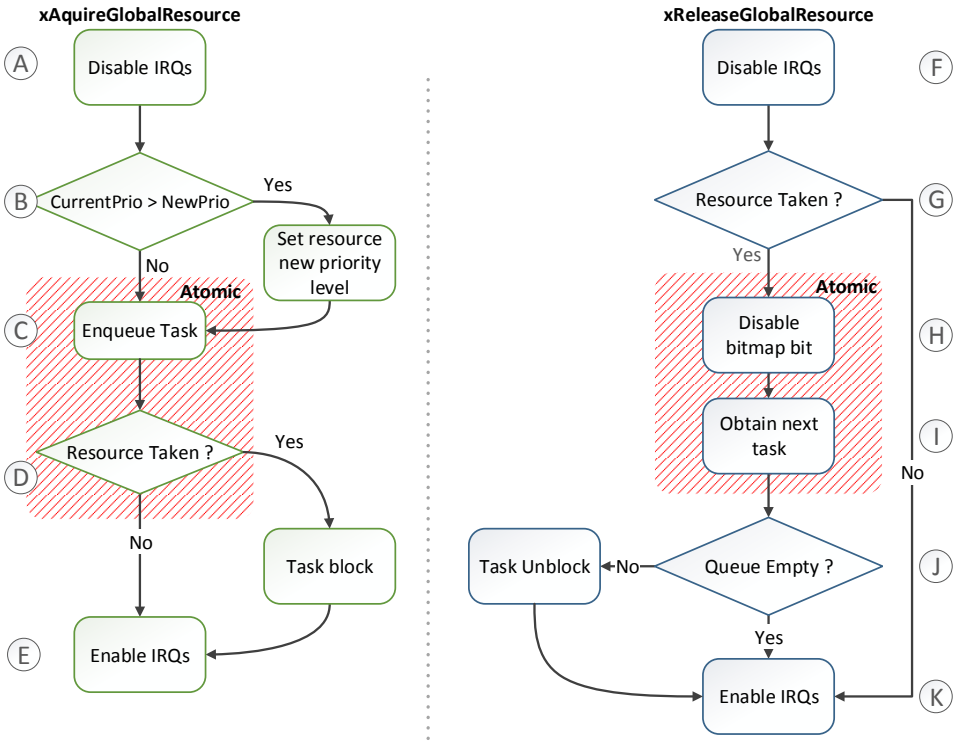


Figure 4.18: Global Resource Acquire and Release

of the task and observing if the resource is currently taken by another task (see Ⓒ and Ⓓ). If the resource is taken the task is then blocked waiting for the resource to be available again, if not, the local interrupts are then enabled and the task acquires the resource Ⓔ.

When releasing a resource, displayed on the right side of figure 4.18, local interrupts are disabled Ⓕ and the following step is to determine if the resource is already taken Ⓖ. The next action is the atomic dequeue of the task and to obtain the next task waiting in the resource queue (see Ⓗ and Ⓘ). If the queue is not empty the highest priority task from the queue is unblocked Ⓙ, and finalizes by enabling local tasks Ⓚ.

4.4.4 ARM Exclusive Access

The most basic synchronization mechanism implemented is not a synchronization mechanism *per si*, but instead are exclusive access directives provided in the ARM instruction set to prevent concurrent memory access to the same memory address. These instructions are ideal to prevent concurrent access to the GIC registers and

synchronization queues, which have the possibility to be accessed simultaneously by more than one core. Two routines were created in order to use this directives to control concurrent access. The first is an exclusive assignment to a register, the `AssignExclusive` function (listing 11) which receives the memory address of the register (or variable) that needs to be accessed, and implements it by using the `LDREX` and `STREX` functions.

```
start_excluse:
    // Load Exclusive
    ldrex r2, [r0]
    orr   r2, r1
    // Store Exclusive with status bit saved in R3
    strex r3, r2, [r0]
    // Check if memory position was altered in the meantime
    teq   r3, #1
    beq   start_excluse
```

Listing 11: Assign Exclusive

The clear exclusive subroutine works in similar fashion to the assign exclusive, but instead of enabling a bit in the GIC's provided register, it clears it, as displayed in listing 12.

```
start_clear_excluse:
    ldrex r3, [r0] // Load Exclusive
    and   r3, r2   // Clear Interrupt Priority
    orr   r3, r1             // Assignment
    strex r4, r3, [r0]       // Store Exclusive with status bit saved ↩
    in R4
    teq   r4, #1             // Check if memory position was altered ↩
    in the meantime
    beq   start_clear_excluse  // If the status bit is one, the load-↩
    store process is re-done
```

Listing 12: Clear Exclusive

Chapter 5

Evaluation

In the previous chapter the implementation was presented, starting with the platform memory model and boot code. Secondly the changes made to the scheduler, to make the operating system use the GIC as its hardware scheduler, as well as the changes to the task related API services to take advantage of the hardware scheduler and the multicore platform running underneath. Finally, the synchronization mechanisms implemented, in order to prevent memory corruption, due to simultaneous memory access to the same memory addresses, with local and global synchronization mechanisms.

In this chapter, the system is evaluated by gathering results from microbenchmarks performed on the ARM Fast Models virtual platform. The same series of tests are performed in the two versions of the FreeRTOS: (i) the native singlecore version with the software scheduler; (ii) the multicore version with the hardware interrupt controller performing the scheduler operations. By running multiple tests with the best and worst case scenarios, it allows the analysis of the performance and determinism of each API service that was refactored. The behaviour of the system is also evaluated in order to check the rate-monotonic priority inversion problems were solved, and if the remaining system behaviour is still the same as in the native version.

5.1 Evaluation Tools

To correctly evaluate the impact of the changes made to the FreeRTOS, performance and determinism metrics were analysed. To do this, each API service needs to be analysed individually. The data gathering process was performed using the performance monitoring unit (PMU).

5.1.1 Performance Monitoring Unit

The Cortex-A9 processor provides the PMU, which in present in both processors.

- The PMU provides six counters to gather processor and memory system data.
- The PMU registers are accessible in the `CP15` interface and from the Debug APB interface.
- The PMU provides each counter with the possibility to count any of the 58 available events in the processor [31].

Table 5.1: API services evaluated with PMU

PMU Registers	**Description**
PMCR	Performance Monitor Control Register
PMCNTENSET	Count Enable Set Register
PMCNTENCLR	Count Enable Clear Register
PMCCNTR	Cycle Count Register

The main registers used in the data gathering process are present in table 5.1. The process starts by enabling the PMU in the `PMCR` register, by setting its least significant bit (reset value is 0x41093000). The next step is to start the counter, which can be done by writing 0x80000000 to the `PMCNTENSET` register before the instruction where the count shall begin. Finally the `PMCCNTR` register then starts counting each clock cycle, after the desired data is gathered the counter can be disabled and reset in the `PMCNTENCLR` register.

5.2 FreeRTOS Evaluation

The FreeRTOS evaluation was performed on two versions of the FreeRTOS, the native singlecore version, executing with a software scheduler and the multicore hardware-centric version of the FreeRTOS. Real-time operating systems require the maximum possible determinism and performance to prevent deadlines from being lost. Consequently, the impact in determinism and performance, in the hardware-based implementation of the FreeRTOS, is the main focus of this evaluation.

Table 5.2 presents all the evaluated API services. The evaluation tests can be divided into two categories: the first, concerns all of the task management API, refactored for the hardware centric version of the FreeRTOS, task creation and deletion, task resume and suspend and changing a priority of a task; The second category is the synchronization mechanisms implemented for local and global shared resources.

Table 5.2: API services evaluated with PMU

API Services	**Description**
`xTaskCreate()`	Create Task
`vTaskDelete()`	Delete Task
`vTaskSuspend()`	Suspend Task
`vTaskResume()`	Resume Suspend Task
`vTaskPrioritySet()`	Change Task Priority
`vSemaphoreCreateBinary()`	Create Resource Mutex
`xSemaphoreTake()`	Take Local Resource
`xSemaphoreGive()`	Give Global Resource
`xSemaphoreGlobalTake()`	Take Global Resource
`xSemaphoreGlobalGive()`	Give Global Resource

5.2.1 Test Scenarios

To correctly evaluate the performance and determinism metrics, specific tests must be executed in both of the previously mentioned versions of the FreeRTOS. Determining performance is fairly straightforward, where the API services are called and the number of clock cycles is counted from the point where the service is called, to the point where it exits. The determinism metric, on the other hand, requires deep understanding of the operating system, since the tests consist of the best case scenarios to obtain the fastest execution times, and also of the worst case scenarios. Where the latter requires knowledge of the operating system inner-workings,

specially how the scheduler works, and what are the main sources that can cause indeterminism, to provide the worst case scenarios.

The FreeRTOS's main sources of indeterminism, in the task related API services, come from the way it performs the context-switching between tasks. When a context-switch occurs, either caused by the task finishing its executing or its forced to yield, by a suspend for instance, the scheduler needs to decided which task should be executed next. The FreeRTOS scheduler does this by running through the task ready list, running through each priority level until it finds a task that is ready to be executed. For instance, figure 5.1 exemplifies a case where this problem is evident: the operating system has two tasks created, one task with a high priority level (255) and another task with a low priority level (2). At this point the scheduler puts the high priority in execution.

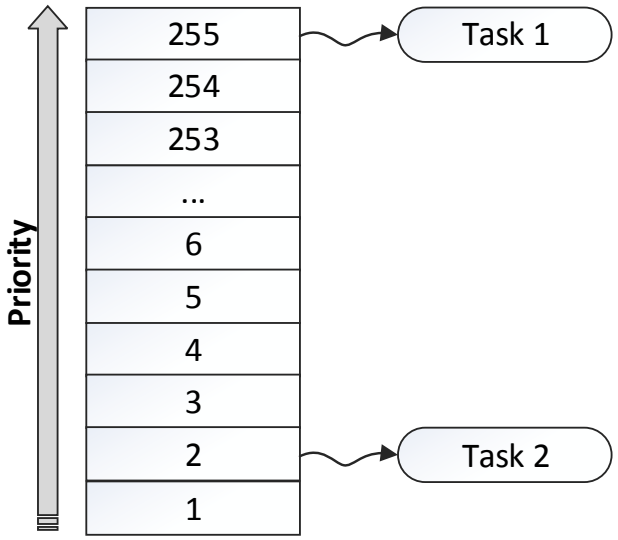


Figure 5.1: Scheduler List for tasks ready to run

If the high priority task is executing, and its execution is suspended or its priority is changed to 1, the lowest possible priority, the scheduler searches through each priority in the ready list, to find a task that is ready to run, as displayed in figure 5.2, which is very time consuming. Furthermore, the time taken in search process, is dependant on the priority level difference between tasks: this case presents the worst case, where the scheduler has to search through every priority level until it finds the next ready-to-run task in the lowest priority level.

Another test that needs to be executed, pertains to the priority inversion issues in

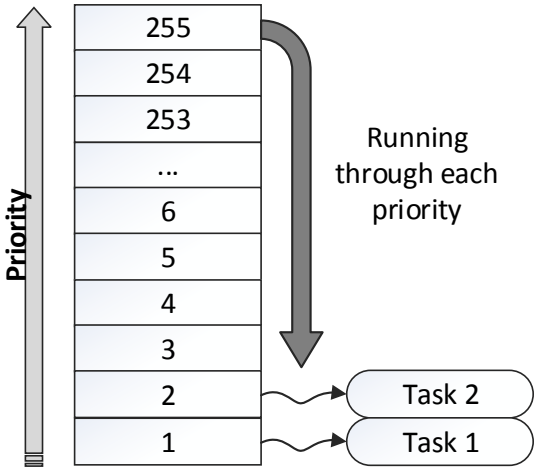


Figure 5.2: Search through scheduler list

the native version of the FreeRTOS. The system begins execution with two tasks created (1 and 3), both suspended, and an IRQ (C). Task A has priority level 1 (lowest), ISR 2 has priority level 3, and task 3 has priority level 4 (highest of the three). The following steps describe execution flow of the native version of the FreeRTOS: (i) Task 1 is unlocked; (ii) ISR 2 is triggered starts executing; (iii) While IRQ C is executing, task 3 is activated, but ISR 2 continues executing; (iv) ISR 2 ends and task 3 begins execution; (v) Task 3 ends, and Task 1 resumes execution; (vi) Task 3 is triggered again and starts executing; (vii) ISR 2 is triggered, and starts executing, despite having lower semantic priority than task 3; (viii) ISR ends execution and task 3 starts executing again.

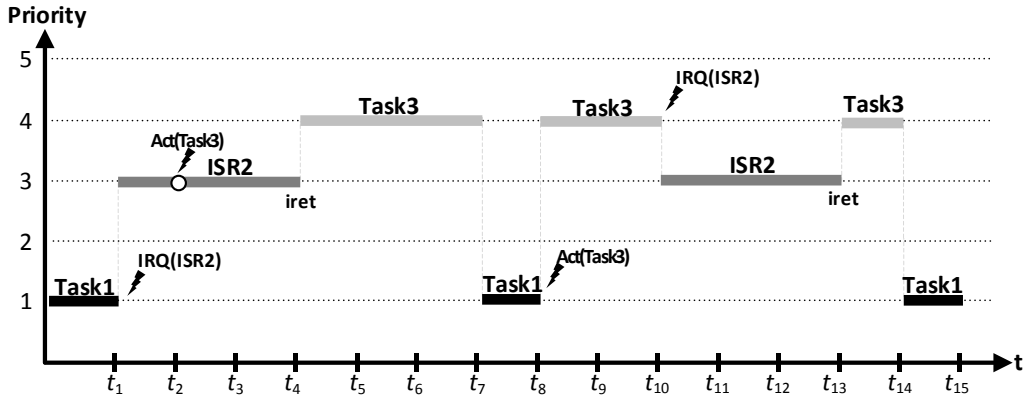


Figure 5.3: Behaviour test case scenario - Bifid Priority Space

This set of events causes task 3, which has the highest priority, to be delayed

multiples times by ISR 2, due to the bifid priority space present in the native version of the FreeRTOS. The desired behaviour is depicted by figure 5.4, where ISR2 will never interrupt task 3, since it has semantically lower priority, thus eliminating the rate-monotonic priority inversion.

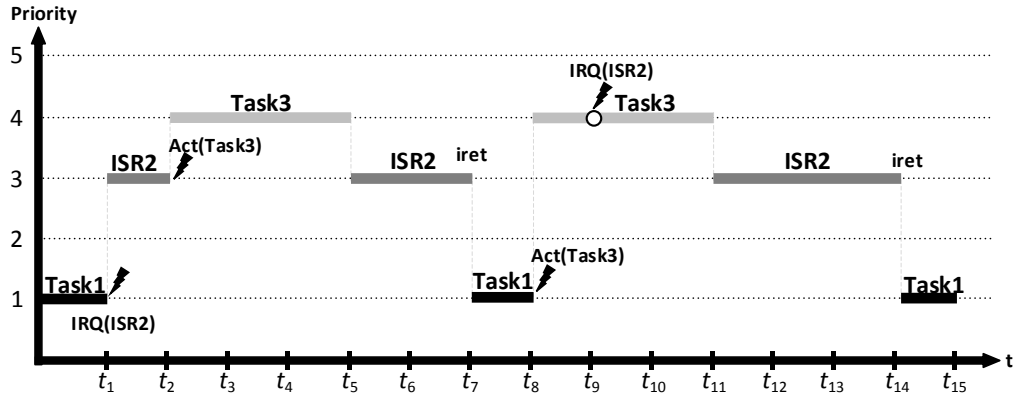


Figure 5.4: Behaviour test case scenario - Unified Priority Space

5.2.2 FreeRTOS Task Management

This subsection describes the performance and determinism evaluation for the task management API services. Comparing the difference in clock cycles that each API service takes, from the point they are called, to the point they exit, in the native version of the FreeRTOS and the newly implemented version. The table sections concerning cross-core interactions are empty in the native FreeRTOS, since the native FreeRTOS targets a singlecore architecture.

xTaskCreate

The `xTaskCreate` was the least impacted among all the tested API services. There were slight performance improvements, as shown in table 5.3. When a task is created, the scheduler automatically knows if the task has higher priority than the one currently executing. Therefore, the `xTaskCreate` doesn't suffer from the aforementioned problem of having to search for the next ready-to-run task. This API service was already deterministic, only a very small fluctuation among the series of tests was found. Consequently, the only impact with this implementation,

was a decrease in the number of instructions executed, a 4.3% decrease in overhead regarding API service calls that trigger a dispatch, and 2.4% in those that don't.

In the cross-core interactions, where a processor core creates a task that targets a different core, there is no difference compared to the intra-core interactions, as displayed in the lower half of the table. Since the *targetcore* variable is assigned either it is a intra-core or cross-core task creation, the process stays absolutely the same. The interrupt controller then takes care of scheduling the task to its target core.

Table 5.3: Performance and determinism evaluation for task create

Dispatch		**Native FreeRTOS**		**HcM FreeRTOS**		
IC	***CC***	$\overline{x}$	s	$\overline{x}$	s	ov.(%)
w	-	1089	2	1042	0	***-4.3***
w/o	-	968	0	945	0	***-2.4***
-	w	-	-	1042	0	**-**
-	w/o	-	-	945	0	**-**

vTaskDelete

The `vTaskDelete` is the first API service analysed that suffers from the problem of very high levels of indeterminism stemming from the aforementioned problem. As displayed in table 5.4, when a `vTaskDelete` triggers a dispatch, meaning the task being deleted is currently executing, the system presents high levels of indeterminism. Test scenarios show that, a `vTaskDelete` can take anywhere from 316, up to 5608 clock cycles to execute. The newly implemented version only takes 306 clock cycles, causing a massive decrease in instruction overhead, 89.6%, and is fully deterministic, since among all the tests there was no fluctuation in the number of clock cycles. Regarding the delete without dispatch, the native version has little fluctuations, and as such there wasn't a big impact in determinism. There were still improvements, since the implemented version presents no fluctuation in clock cycles and there was also an improvement in performance since it takes less 19 clock cycles (10.2% decrease in overhead) to perform the `vTaskDelete` without dispatch.

Cross-core activations of the `vTaskDelete` that trigger a dispatch, take more clock cycles to execute than intra-core activations due to the need for cross-core communication, using software generated interrupts as explained in subsection 4.3.7. This

impact is translated into an increase of 265 clock cycles for cross-core activations of the task delete API service. Regarding the delete without dispatch, the number of clock cycles remains unaffected, since the task is marked as waiting deletion, and when the core containing the task that was marked to be deleted enters the idle function, it then frees the task memory, without the need to signalled the core using cross-core communication.

Table 5.4: Performance and determinism evaluation for task delete

Dispatch		**Native FreeRTOS**		**HcM FreeRTOS**		
IC	***CC***	$\overline{x}$	s	$\overline{x}$	s	ov.(%)
w	-	2955	2891	306	0	***-89.6***
w/o	-	187	17	168	0	***-10.2***
-	w	-	-	571	0	**-**
-	w/o	-	-	168	0	**-**

vTaskSuspend and vTaskResume

The `vTaskSuspend` suffered great improvements with the new implementation. Similarly to the `vTaskDelete`, there was a significant increase in performance and determinism, as displayed in table 5.5. In the native version of the FreeRTOS, when the API service triggers a dispatch, the suspend process can take anywhere from 302 (best case scenario) to 5594 (worst case scenario) clock cycles. The implemented approach to the `vTaskSuspend` API service takes only a constant 283 clock cycles to execute, when considering intra-core operations, and 446 when the task being suspended is executing on another processor core (cross-core suspension that triggers a dispatch). This variation happens due to the need for interprocessor communication, to signal the other processor core that it needs to suspend the currently executing task. The API service call which do not trigger a context-switch had already very little fluctuations in the number of clock cycles in the native version, and once again in this implementation the performance was increased, reflected in a decrease of nearly 50%. The aforementioned variation is now zero, since suspending a task that is not executing, consists only in disabling it's corresponding interrupt pending bit.

Although the original `vTaskResume` has deterministic behaviour, with only a negligible 2 clock cycle variation among all performed test scenarios, this approach greatly increased it's performance (table 5.6). While maintaining a deterministic

Table 5.5: Performance and determinism evaluation for task suspend

Dispatch		**Native FreeRTOS**		**HcM FreeRTOS**		
IC	***CC***	$\overline{x}$	s	$\overline{x}$	s	ov.(%)
w	-	2941	2891	283	0	***-90.4***
w/o	-	158	2	81	0	***-48.6***
-	w	-	-	446	0	**-**
-	w/o	-	-	81	0	**-**

implementation, the approach presented in this dissertation presents a 49.6% decrease in overhead when the resume API service triggers a context switch, and a 75.4% decrease in overhead when a dispatch does not occur. Regarding cross-core activations of the `vTaskResume` API service, the number of clock cycles remains exactly the same as the one concerning intra-core activations. Since the resume consists of enabling the pending bit corresponding to the suspended task-associated interrupt, there is no need for cross-core communication. The hardware interrupt controller takes care of scheduling the resumed task, if its priority level warrants the preemption of the currently executing task on its target processor core.

Table 5.6: Performance and determinism evaluation for task resume

Dispatch		**Native FreeRTOS**		**HcM FreeRTOS**		
IC	***CC***	$\overline{x}$	s	$\overline{x}$	s	ov.(%)
w	-	301	2	152	0	***-49.6***
w/o	-	207	0	51	0	***-75.4***
-	w	-	-	152	0	**-**
-	w/o	-	-	51	0	**-**

vTaskPrioritySet

The `vTaskPrioritySet` was the last analysed task management API service. In the native version of the FreeRTOS, when the API service triggers a dispatch, there is a great variation in the number of clock cycles taken to perform a change in priority. Since the newly implemented version only needs to change the priority in the GIC associated register, it only takes a constant 170 clock cycles to perform the same job and is deterministic, leading to a 94.3% decrease in overhead. As previously mentioned the only action needed is to change the priority in the GICs register, and since the GIC takes care of triggering the task-associated interrupt

on any given core, there is no need for cross-core communication mechanisms. Consequently the number of clock cycles needed to perform a change in priority remains unaffected regardless of if it's a core changing the priority of a task in another core, or in a task running on the same core.

Table 5.7: Performance and determinism evaluation for task priority change

Dispatch		**Native FreeRTOS**		**HcM FreeRTOS**		
IC	***CC***	$\overline{x}$	s	$\overline{x}$	s	ov.(%)
w	-	2976	2884	170	0	***-94.3***
w/o	-	278	64	74	0	***-73.3***
-	w	-	-	170	0	**-**
-	w/o	-	-	74	0	**-**

5.2.3 FreeRTOS Synchronization Mechanisms

In this subsection the synchronization mechanisms are characterized. The creation of a semaphore is the first characterized API, where the native API is compared with the newly refactored creation of global and local semaphores. The next step is the characterization of the local and global acquiring and relinquishing access to a resource. Lastly, behaviour tests are performed, in order to verify if the rate-monotonic priority inversion and deadlocks that are built-in the native FreeRTOS are solved with this approach to the synchronization mechanisms.

vSemaphoreCreateBinary

The semaphore creation system service is different from all other tested, in the sense that it does not cause dispatching of any task. The results show that the creation of a semaphore is deterministic in both native and hardware-centric versions of the FreeRTOS, yet, the latter has a better performance. Translated in a decrease in the number of clock cycles from 675 to 543 from to native the hardware-centric version, as displayed in table 5.8, which means a decrease in performance overhead of 19.5% .

Table 5.8: Performance and determinism evaluation for semaphore creation

vSemaphoreCreateBinary	$\overline{x}$	***s***
Native FreeRTOS	675	0
HcM FreeRTOS (Local)	543	0
HcM FreeRTOS (Global)	543	0

xSemaphoreTake

Acquiring a resource, in the native version of the FreeRTOS, presents a clear source of indeterminism, as shown in table 5.9. A task which attempts to acquire an already taken resource, will trigger a context switch from the running task to the task which holds the resource. This context-switch, as previously mentioned, leads to high levels of indeterminism. For this reason, tests performed show that, a `xSemaphoreTake`, which triggers a context-switch, takes a minimum of 821, and a maximum of 6113 clock cycles to perform. The implementation presented in this dissertation, not only solves the indeterminism issues, but also increases performance. In API calls that trigger a context-switch, there is an decrease of 93.1% in performance overhead, and a decrease of 13.6% in those that don't.

Table 5.9: Performance and determinism evaluation for local semaphore take

Dispatch	**Native FreeRTOS**		**HcM FreeRTOS**		
	$\overline{x}$	s	$\overline{x}$	s	ov.(%)
w	3467	2646	238	0	***-93.1***
w/o	140	0	121	0	***-13.6***

xSemaphoreGive

Releasing a resource, regarding the native version of the FreeRTOS, was already deterministic. All performed test scenarios presented no clock cycle variation, as shown in table 5.12. The approach presented in this dissertation to the implementation of local synchronization mechanisms, in particular the `xSemaphoreGive`, maintains the deterministic behaviour of the native version, while also improving performance. This performance increase is translated in a decrease in performance overhead of 32.7% in the API service which triggers a context-switch, and 6.7% in those who do not trigger a context-switch.

Table 5.10: Performance and determinism evaluation for local semaphore give

Dispatch	**Native FreeRTOS**		**HcM FreeRTOS**		
	$\overline{x}$	s	$\overline{x}$	s	ov.(%)
w	486	0	327	0	***-32.7***
w/o	104	0	97	0	***-6.7***

xSemaphoreGlobalTake

The global acquiring of a resource has similar performance, when compared to the local `xSemaphoreTake`, maintaining the deterministic behaviour of the local resource acquiring. However there is a slight decrease in performance when compared to the local version, since there is a need to perform the atomic enqueue of tasks attempting to acquire the resource. Table 5.11 presents the comparison with the native FreeRTOS version, which is still characterized with a decrease, although slightly smaller, in performance overhead of 93.1% regarding the system calls that trigger a context-switch and 13.6% for those who do not.

Table 5.11: Performance and determinism evaluation for global semaphore take

Dispatch		**Native FreeRTOS**		**HcM FreeRTOS**		
IC	***CC***	$\overline{x}$	s	$\overline{x}$	s	ov.(%)
w	-	3467	2646	247	0	***-93.1***
w/o	-	140	0	130	0	***-13.6***
-	w	-	-	247	0	**-**
-	w/o	-	-	130	0	**-**

xSemaphoreGlobalGive

The `xSemaphoreGlobalGive` system service performance evaluation results are very similar to the `xSemaphoreGive` (local resource acquiring). The global relinquishing of a resource maintains the deterministic behaviour of the local resource acquiring, but displays a slight performance decrease, due to the atomic queue operations. As displayed in table 5.12, this system service, when compared to the native FreeRTOS, presents a decrease in the performance overhead of 90.5%, regarding the relinquishing of a resource which triggers a context, and a slight increase of 3.4%, concerning those who do not trigger an interrupt.

Table 5.12: Performance and determinism evaluation for global semaphore give

Dispatch		**Native FreeRTOS**		**HcM FreeRTOS**		
IC	***CC***	$\overline{x}$	s	$\overline{x}$	s	ov.(%)
w	-	486	0	328	0	***-90.5***
w/o	-	104	0	108	0	***+3.4***
-	w	-	-	328	0	**-**
-	w/o	-	-	108	0	**-**

Behaviour Test

Aside from the performance and determinism tests, the behaviour of the system needs to be evaluated, to confirm the rate-monotonic priority inversion issues, inherent to the native version of the FreeRTOS, are solved. Figure 5.5 displays the results behaviour test performed on the hardware-centric FreeRTOS. Contrary to the behaviour of the native FreeRTOS (presented in figure 5.3), in this approach, when task 3 is activated (t2), ISR 2 stops execution, and task 3, which has higher priority, starts executing. Similarly, when ISR2 is triggered (t9), task 3 continues its execution, since its priority is higher than the ISR. This approach follows Therefore, this approach dissolves the rate-monotonic priority inversion present in the native FreeRTOS.

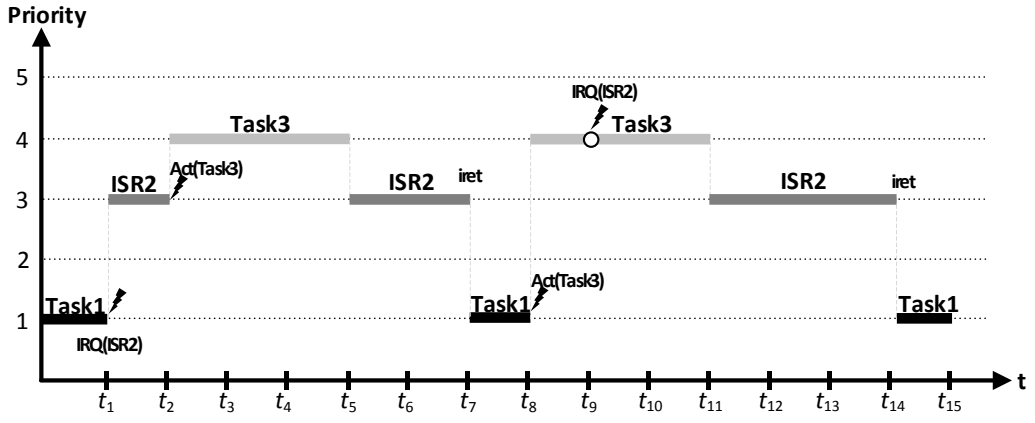


Figure 5.5: Behaviour test - Hardware-centric FreeRTOS

5.3 Overall Evaluation

The overall evaluation results for the task management API services are presented in figure 5.6. The aforementioned figure shows the comparative results of each API service with their respective variation for the native version of the FreeRTOS and the hardware-centric multicore version, where a distinction is made between the intra-core and cross-core interactions. As displayed, the indeterminism issues inherent in the native version were greatly diminished in the hardware-based version, and the performance of all API services is also improved. The variation of clock cycles displayed in hardware-centric FreeRTOS API services is related to the context-switch operation, specially when cross-core communication mechanisms are required.

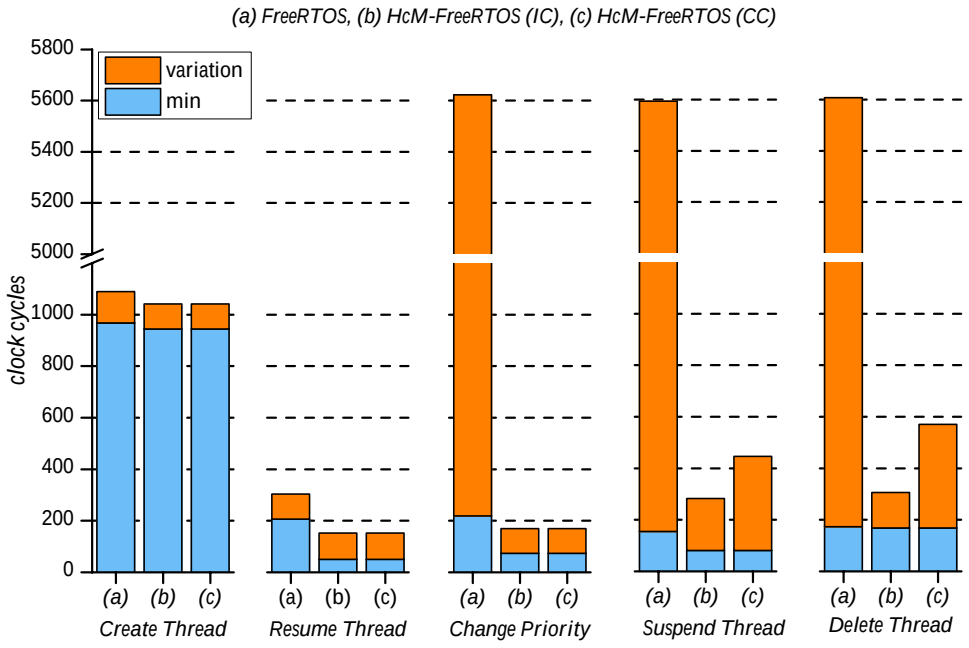


Figure 5.6: Overall results for task management API services

The results for the synchronization mechanisms are presented in figure 5.7. The aforementioned figure highlights the indeterminism present in the native version of the FreeRTOS synchronization mechanisms, specially in the acquiring of a resource due to the context-switching operations. The synchronization mechanisms implemented in this dissertation, not only greatly improve overall determinism,

but also improve performance across all refactored system calls.

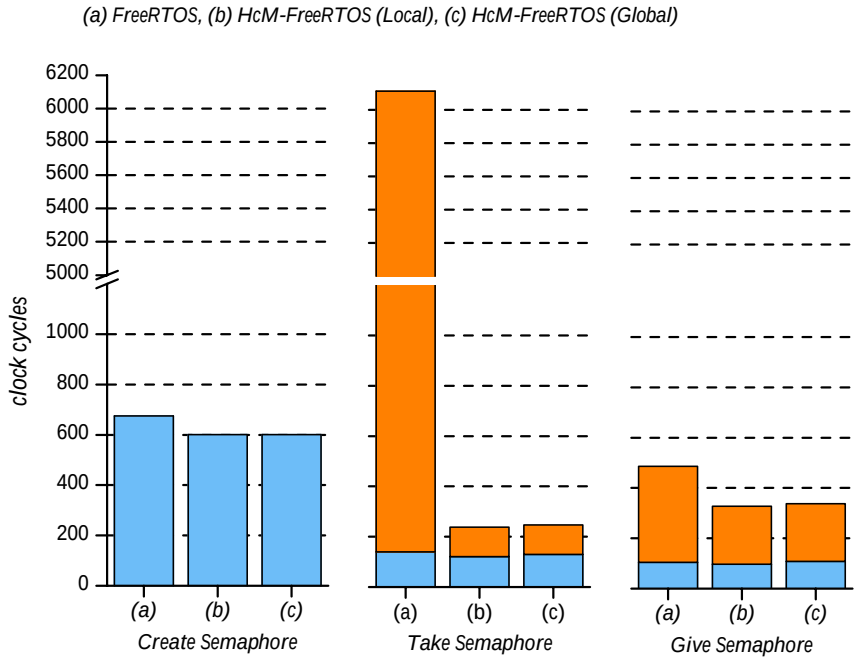


Figure 5.7: Overall results for synchronization API services

Chapter 6

Conclusion and Outlook

In this last chapter, conclusions regarding the work developed and the advantages of a more hardware-centric approach to real-time operating systems are presented. Furthermore, suggestions to future work are discussed.

6.1 Summary and Conclusions

This dissertation presented the implementation of a multicore hardware-centric version of the FreeRTOS, by offloading some kernel components to COTS hardware. The main focus is the advantages of a more hardware-centric approach to real-time operating systems, where the optimization of determinism and efficiency in operating system properties are crucial so that deadlines can be met.

This was without a doubt a very challenging project, due to its complexity and the embedded systems field knowledge required: understanding the processor and the generic interrupt controller architectures, figuring out the differences in multicore architectures, their advantages and limitations regarding real-time operating systems, assembly language, deep knowledge of the FreeRTOS inner-workings and synchronization mechanisms, regarding multicore operating systems. All of the aforementioned concepts were essential to successfully complete this dissertation.

Overall, the proposed objectives for this dissertation were met. The migration of the software scheduler to the hardware interrupt controller and subsequent refactoring of the task management API, to provide support for the hardware scheduler, was successfully implemented. The operating system was extended to an

hardware-based symmetric multiprocessing architecture, which was implemented on a dual-core platform. However it can easily be extended to operate with an higher number of cores, for instance, quad-core. The synchronization mechanisms were successfully implemented, allowing both local and global resource management. Finally, the last objective was the analysis of system metrics, to determine the impact in both determinism and performance of the operating system when compared to the native version of the FreeRTOS. The collected results corroborate the advantages of a more hardware-centric approach, taking advantage of the ever more powerful hardware subsystem.

6.2 Future Work

Despite all the predetermined objectives being met, there are still many ways to improve this approach. Future work suggestions targets multiple fronts.

The first suggestion for future work is related to the implementation of a time-out on task blocking, regarding the synchronization mechanisms. The native version of the `xSemaphoreTake` API allows the specification of the maximum amount of time a task should remain in the blocked state, by specifying the number of the tick periods in the *xTicksToWait* API parameter. On the other hand, this implementation does not have this feature, only allowing an indefinite amount of waiting time.

The second suggestion is related to the implementation of a more complex load balancing algorithm. One of the most challenging ways to increase performance in multicore architectures is the ability to do so, without programmer intervention. This implementation uses a round-robin schema to distribute tasks among cores and an API service for task core affinity was implemented. However, round-robin algorithms can lead to a load-imbalance (uneven CPU utilization).

The third suggestion is an in-depth and real-world system evaluation. Performing system evaluation in a physical multicore development platform (e.g., Xiling Zyqn ZC702) to assess real-world results. Although Fast Models is an excellent tool for proof-of-concept, it does not model accurate cycle counts, since all instructions take the same amount of time. Memory footprint and code management (maintainability) are other metrics which are worth being evaluated.

From a different perspective, the next possibility is the development of an effi-

cient hardware-based dual-OS architecture. With the increasing complexity of modern embedded devices, which increasingly demand general purpose computing characteristics, but still need to guarantee real-time requirements to develop efficient solutions that allow the coexistence of General Purpose Operating Systems (GPOSes) with RTOSes. The solution presented by Sandro Pinto et. al. in "*Towards a lightweight embedded virtualization architecture exploiting arm trustzone*" [34], provides the aforementioned need for spatial and temporal isolation between the OSes.

Appendix A

HcM-FreeRTOS Article

During the development of this dissertation, an work-in-progress (WIP) paper was published and presented in the Emerging Technologies and Factory Automation (ETFA), 20th IEEE International Conference, named "*HcM-FreeRTOS: Hardware-centric FreeRTOS for ARM Multicore*". This paper presented the implementation of the multicore version of the FreeRTOS with the offloaded scheduler to the hardware interrupt controller, without the synchronization mechanisms yet implemented. Preliminary results are presented concerning the differences in performance and determinism achieved with this implementation.

HcM-FreeRTOS: Hardware-centric FreeRTOS for ARM Multicore

E. Qaralleh*, D. Lima**, T. Gomes**, A. Tavares**, S. Pinto**
*Princess Sumaya University for Technology
**Centro Algoritmi - University of Minho
qaralleh@psut.edu.jo
{diogo.lima, mr.gomes, atavares, sandro.pinto}@dei.uminho.pt

***Abstract*—Migration to multicore is inevitable. To harness the potential of this technology, embedded system designers need to have available operating systems (OSes) with built-in capabilities for multicore hardware. When designed to meet real-time requirements, multicore SMP (Symmetric Multiprocessing) OSes not only face the inherent problem of concurrent access to shared kernel resources, but still suffer from a bifid priority space, dictated by the co-existence of threads and interrupts.**

This work in progress paper presents the offloading of the FreeRTOS kernel components to a commercial-off-the-shelf (COTS) multicore hardware. The ARM Generic Interrupt Controller (GIC) is exploited to implement a multicore hardware-centric version of the FreeRTOS that not only solves the priority inversion problem, but also removes the need of internal software synchronization points. Promising preliminary results on performance and determinism are presented, and the research roadmap is discussed.

***Index Terms*—Unified Priority Space, RTOS, Multicore, Real-time Systems, FreeRTOS, GIC, Cortex-A9 MPCore, ARM.**

I. Introduction

Multicore technology has proven to be the only viable solution to achieve high performance without compromising power consumption [1], and its use on desktops and server environments is now ubiquitous. Over the last few years, the use of multicore processors in the embedded systems field has been growing rapidly [2], driven by the lack of performance in single-core processors to meet the demands of the current software-rich generation of embedded devices. However, in order to harness the potential of this technology, embedded system designers need to deploy applications under embedded operating systems (OSes) with built-in support for available multicore hardware [3].

OSes, in general, suffer from a bifid priority space. Threads, managed by software (kernel scheduler), need to coexist with interrupt service routines (ISRs), managed by hardware (interrupt controller). This division into thread priorities and interrupt priorities, with the latter having a higher privilege of execution, is critical on embedded real-time systems, originating a well-identified problem known as rate-monotonic priority inversion [4]. Kleyman and Eykholt have proposed the first solution [5] many years ago, and since then many other approaches have been proposed to tackle this problem [6], [7], [8], [9].

Multicore OSes, for instance, still face another problem. When designed for symmetric multiprocessing, they are conceived to satisfy two principles: (i) minimize the memory footprint; and (ii) have a full and homogeneous utilization of the processor resources. However, since the kernel data structures are present in shared memory, synchronization mechanisms for concurrent access need to be introduced internally [10]. These internal software synchronization points constitute a considerable source of indeterminism, becoming the main reason why embedded real-time operating systems (RTOSes) are delaying their transition to multicore. Only recently Müller et al. [11] addressed both aforementioned problems, extending the philosophy of the SLOTH [7] concept to the multicore domain, implemented over the AUTOSAR OS and targeting the Infineon AURIX platform.

This work in progress paper presents the implementation of a multicore hardware-centric version of the FreeRTOS, by offloading critical run-time kernel services to commercial-off-the-shelf (COTS) hardware. By exploiting the GIC of the ARM Cortex-A9 MPCore to migrate FreeRTOS system services to hardware, not only the priority inversion problem is solved, but also the need for internal software synchronization points is removed. The thread-related application programming interface (API) was kept syntactically intact to avoid the porting effort for legacy applications. Preliminary results have shown significant improvements in overall system performance and determinism.

II. Design of HcM-FreeRTOS

The rationale behind HcM-FreeRTOS is representing tasks and ISRs by abstract interrupt sources, configuring its priority and target core (Fig. 1). The system consists of (i) task activation, (ii) task dispatching and (iii) task suspending. Synchronous task activation relies on triggering the associated interrupt source via software, by writing on its respective interrupt controller register. Task dispatching, in turn, is based on saving the context (not implicitly saved by hardware) of the current executing task, followed by a branch to the new highest priority ready-to-run task. Finally, task suspending consists in forcing a task to yield its execution flow, allowing other lower priority tasks/ISRs to run. Since it is not intrinsically supported by hardware, it requires a more complex IRQ handler, which will save the context of the currently executing task in a

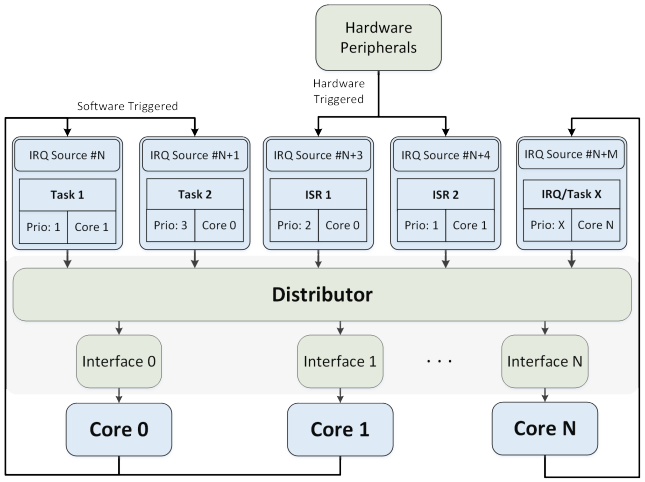


Fig. 1: Design of the hardware-centric multicore system, using interrupt handlers for the implementation of threads

dedicated stack, disable the interrupt source, and dispatch (resuming/restoring) the next ready-to-run task/ISR.

Targeting multicore platforms, HcM-FreeRTOS needs to support task activation, dispatching and suspending across multiple cores. To accomplish that, the aforementioned building blocks should be extended with a cross-core communication mechanism, typically available in the form of interprocessor interrupts.

A. Hardware Requirements

The implementation of this hardware-centric approach is only feasible if the underlying multicore hardware platform fulfils certain requirements: (i) the hardware interrupt controller must be programmable and provide several different configurable interrupt priority levels; (ii) the interrupt subsystem shall support manual triggering of interrupts as well as by software, enabling threads to be synchronously activated; (iii) the sum of the number of interrupt sources and priorities per core should be enough to cover all the threads and interrupt handlers desired for the system; (iv) a special instruction or mechanism should exist to send interrupts to remote cores.

III. Implementation of HcM-FreeRTOS

This section presents an overview of the ARM interrupt controller subsystem - GIC -, as well as a description about the HcM-FreeRTOS implementation.

A. ARM Generic Interrupt Controller

One of the COTS interrupt subsystems that fulfils the aforementioned hardware requirements is the ARM GIC, integrated in all multicore ARM Cortex-A System-on-Chips. It is partitioned into two logical blocks: the distributor and the CPU interface. The former determines the highest priority interrupt for each core and dispatches them to each CPU interface, while the latter is responsible for handling the arbitration of incoming interrupt requests locally. The GIC provides up to 1023 interrupt sources, classified in three different categories: (i) SGIs (Software Generated Interrupts) (0-15) - special interrupts generated by software for interprocessor interrupts, banked for all cores; (ii) PPIs (Private Peripheral Interrupts) (16-31) - peripheral interrupts specific to a single processor, banked for all cores; (iii) SPIs (Shared Peripheral Interrupts) (32-1023) - general interrupts shared among all cores.

B. Threads as Interrupts as Threads

As mentioned in Section II the main idea behind HcM-FreeRTOS is designing software tasks as hardware interrupts. However, the main drawback of having tasks run as pure interrupts is the run-to-completion nature of the hardware interrupt handlers. To overcome this limitation and extend interrupts to behave also as threads, a suspending feature was implemented by modelling tasks as consisting of three segments: prologue, body and epilogue [8]:

- *Prologue*: The prologue is executed wherever a high priority task is scheduled by the interrupt controller. It extends the standard behaviour of the hardware interrupt controller (i.e., it saves automatically some registers of the CPU context) by saving the remaining context of the current task, and restoring the context of the new task.
- *Body*: The body implements the task behaviour and corresponds to the application written to run in the native version of the FreeRTOS.
- *Epilogue*: The epilogue is executed wherever a task is suspended or finished. If the task was suspended, it saves the tasks context and then restores the state of the new dispatched task, otherwise the task was finished and it only restores the context of the new task.

The remaining of this section describes how the FreeRTOS task-specific system calls were re-factored. Since the ARM instruction set provides dedicated instructions that allow read and write memory atomically, no additional software synchronization points were necessary to include in order to deal with concurrent access to the GIC distributor registers.

1) Scheduler Start: Starting the scheduler is fairly straightforward, consisting in enabling the GIC distributor and each CPU interface through the `GICD_CTRL` and `GICC_CTRL` registers, respectively.

2) Task Creation/Activation: Whenever a task is created, the existing TCB structure is initialized (allocating the task stack), and the associated interrupt is configured. Thereby, the GIC distributor requires specifying the priority level (`GICD_IPRIORITYRx`), setting the target CPU (`GICD_ITARGETSRx`) - following a round robin schema -, linking the interrupt handler to the task-specific code, enabling (`GICD_ISENABLERx`) and setting the interrupt as pending (`GICD_ISPENDRx`). After, locally to each CPU interface, if the created task has higher priority than the currently executing task, the prologue is executed and the created task dispatched, hence no SGI (i.e., cross-core interaction) is needed.

3) Task Deletion: Whenever a task is deleted the interrupt source linked to that task is disabled (`GICD_ICENABLERx`), it is signalled as waiting deletion (to free the memory during the idle periods), and if the task is currently executing in the local

core then the epilogue is performed, otherwise an SGI is sent (i.e., cross-core interaction) to signal the remote core to delete the current task.

4) Task Suspend: Whenever a task is suspended the pending flag of the interrupt source linked to that task is disabled (`GICD_ICPENDRx`), and if the task is currently executing in the local core the epilogue is performed, otherwise an SGI is sent (i.e., cross-core interaction) to signal the remote core to suspend the current task.

5) Task Resume: Whenever a task is resumed its state is changed to ready and the interrupt is set as pending (`GICD_ISPENDRx`). After, locally to each CPU interface, if the resumed task has higher priority than the currently executing task, the prologue is executed and the resumed task dispatched.

IV. Preliminary Results

The implemented solution was tested on the Fast Models emulator, using a model of the Versatile Express (VE) board with a dual- and quad-core ARM Cortex-A9. We compared the native single-core version of the FreeRTOS (ver. 7.0.2) against our redesigned hardware-centric multicore version (HcM-FreeRTOS). For our implementation we experimented and gathered results also for cross-core interaction. The results were obtained using the Performance Monitoring Unit (PMU) component, and the software was compiled with the ARM Xilinx Toolchain.

In order to assess the performance and determinism results we performed several microbenchmarks. The selected microbenchmarks encompass the modified system calls, which include:

- `xTaskCreate`: Creates a task and dispatches it if its priority is higher than the currently executing task;
- `vTaskDelete`: Deletes a task and dispatches another if the deleted task is currently executing;
- `vTaskSuspend`: Suspends a task and dispatches another if the suspended task is currently executing;
- `vTaskResume`: Resumes a task and dispatches it if its priority is higher than the currently executing task;
- `vTaskSetPriority`: Changes the priority of a task and dispatches it if the modified priority is higher than the priority of the currently executing task;

For each microbenchmark we performed several experiments with different system configurations, changing parameters such as: (i) the number of tasks (from 1 to 32); (ii) the priority of tasks (from 1 to 255); (iii) the number of tasks with the same priority (from 1 to 3); (iv) the priority gap between tasks (from 32 to 253), and (v) the priority of the dispatched or not dispatched task. The presented results report the mean value ($\overline{x}$) and the standard deviation (s) of a set of experiments.

We start by performing the behaviour evaluation, by extending the aforementioned test scenarios with distinct priority levels of hardware interrupts. It is naturally perceptible that we solved the rate monotonic priority inversion problem by design, and we effectively corroborated our predictions by observing an unified execution flow, with tasks and ISRs coexisting correctly. During all experiments no ISR with semantically low priority has interrupted a task with higher priority, fact that was not observed in the native version of FreeRTOS.

	Dispatch		**FreeRTOS**		**HcM**		
API	***IC***	***CC***	$\overline{x}$	***s***	$\overline{x}$	***s***	***ov.(%)***
xTaskCreate	w	-	1089	2	1042	0	***-4.3***
	w/o	-	968	0	945	0	***-2.4***
	-	w	-	-	1042	0	**-**
	-	w/o	-	-	945	0	**-**
vTaskDelete	w	-	2955	2891	306	0	***-89.6***
	w/o	-	187	17	168	0	***-10.2***
	-	w	-	-	571	0	**-**
	-	w/o	-	-	168	0	**-**
vTaskSuspend	w	-	2941	2891	283	0	***-90.4***
	w/o	-	158	2	81	0	***-48.6***
	-	w	-	-	446	0	**-**
	-	w/o	-	-	81	0	**-**
vTaskResume	w	-	301	2	152	0	***-49.6***
	w/o	-	207	0	51	0	***-75.4***
	-	w	-	-	152	0	**-**
	-	w/o	-	-	51	0	**-**
vTaskPrioritySet	w	-	2976	2884	170	0	***-94.3***
	w/o	-	278	64	74	0	***-73.3***
	-	w	-	-	170	0	**-**
	-	w/o	-	-	74	0	**-**

TABLE I: Performance and Determinism Evaluation Results

Table I, in turn, presents the achieved results for the overhead evaluation. It is clear that our implementation overcomes the native version of FreeRTOS in both metrics: performance and determinism. Relatively to the former, the execution time was reduced between 2.4% (`xTaskCreate` without dispatch) and 94.3% (`vTaskPrioritySet` with dispatch). The speedup achieved in the `xTaskCreate` API is considerably smaller than the other APIs, because we implemented the suspend feature and so, it still requires stack allocation for each task. Regarding determinism, our evaluation methodology outlined important sources of indeterminism in FreeRTOS, stemming from its searching algorithm - that determines the next running task - in the context switch operation. Since our approach is based on hardware and relies on the GIC to provide the highest priority ready-to-run task, the context switch, in particular, and the APIs, in general, are naturally deterministic. This is why our approach achieved a null standard deviation in all experiments.

The results still corroborated the viability of implementing multicore RTOS without the need of software synchronization mechanisms for the concurrent access to shared kernel resources. We only needed to guarantee the coherency during GIC distributor registers accesses, and we did that with specific and dedicated hardware instructions. Determinism was not compromised and the only extra overhead with the advent of multicore migration came from cross-core interaction, due to the need to trigger interprocessor interrrupts (SGIs).

V. Research Roadmap

Work in the near future will proceed through the migration of the remaining kernel services to hardware. At this stage, only a subset of the task management API is exploiting the hardware interrupt controller to implement the scheduling decisions. However, the idea is not only offload to hardware all the scheduling services, but also implement other kernel services. For example, synchronization mechanisms such as mutual exclusion (mutex), can easily be implemented exploiting the GIC. For local resources (resources shared between the same processor) the Priority Ceiling Protocol based on the temporary raise of tasks' priority will be implemented [8]. For global resources (resources shared between different processors), we will investigate the applicability of the Multiprocessor Priority Ceiling Protocol [12].

After migrating all kernel services, research will focus on an in-depth and real-world system evaluation. First, we will focus on performance and indeterminism. All the kernel services will be evaluated, not only by performing microbenchmarks, but also running concrete benchmark suites - Thread Metrics and MiBench Suites are good candidates. More sources of indeterminism will be also investigated, and characterized. Furthermore, experiments will be carried out in a physical multicore development platform (e.g., Xilinx Zynq ZC702) to assess real-world results, because Fast Models is an excellent tool for proof-of-concept but does not model accurate cycle counts - all instructions take the same amount of time. Memory footprint and code management (maintainability) are other metrics that will be also evaluated.

From a different perspective, research will continue towards the development of an efficient hardware-based dual-OS architecture. With the emergent complexity of modern embedded devices, which increasingly demand for general purpose computing characteristics but still need to guarantee the real-time requirements, it is necessary to develop efficient solutions that allow the coexistence of General Purpose Operating Systems (GPOSes) with RTOSes. Thereby, the solution of our previous work with ARM TrustZone technology [13] will be followed to provide the spatial and temporal isolation between the OSes. Moreover, we will go one step further applying concepts of this work to the RTOS running on the secure side. By exploiting only ARM COTS hardware, we will provide efficiency and optimization at two different levels of the system stack: not only on the virtualization layer but also in the OS layer.

VI. Conclusion

Over the last few years, the interest in embedded multicore systems has increased significantly due to the simultaneous advantages on power and performance. However, embedded RTOSes with built-in multicore support face two well identified problems: the need of internal software synchronization points, and the lack of an unified priority space. This paper presented a work in progress towards the implementation of a multicore hardware-centric version of the FreeRTOS, by offloading some kernel components to COTS hardware. Migrating the scheduling decisions to the interrupt controller we showed that it is possible to overcome the problems of multicore RTOSes and simultaneously improve performance and specially determinism.

The research roadmap section described that research in the near future will focus on the migration of the remaining kernel services to hardware, and on an extensive system evaluation on a real multicore platform. Research will then proceed towards the development of an efficient hardware-centric dual-OS architecture, by exploiting only COTS ARM SoC capabilities.

VII. Acknowledgement

Sandro Pinto is supported by FCT - Fundação para a Ciência e Tecnologia (grant SFRH/BD/91530/2012). This work has been supported by FCT - Fundação para a Ciência e Tecnologia within the Project Scope: PEst-UID/CEC/00319/2013.

References

[1] M. Karlsson, "Enea Hypervisor: Facilitating Multicore Migration with the Enea Hypervisor," *ENEA White Paper*, pp. 1–11, 2012.

[2] F. Reichenbach and A. Wold, "Multi-core technology – next evolution step in safety critical systems for industrial applications?" in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, Sept 2010, pp. 339–346.

[3] D. Andrews, I. Bate, T. Nolte, C. Otero-Perez, and S. M. Petters, "Impact of embedded systems evolution on rtos use and design," in *1st International Workshop Operating System Platforms for Embedded Real-Time Applications (OSPERT'05)*, 2005.

[4] L. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz, "Predictable interrupt management for real time kernels over conventional pc hardware," in *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, April 2006, pp. 14–23.

[5] S. Kleiman and J. Eykholt, "Interrupts as threads," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 2, pp. 21–26, Apr. 1995.

[6] L. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz, "Predictable interrupt scheduling with low overhead for real-time kernels," in *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, 2006, pp. 385–394.

[7] W. Hofer, D. Lohmann, and W. Schroder-Preikschat, "Sleepy sloth: Threads as interrupts as threads," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, Nov 2011, pp. 67–77.

[8] S. Pinto, J. Pereira, D. Oliveira, F. Alves, E. Qaralleh, M. Ekpanyapong, J. Cabral, and A. Tavares, "Porting sloth system to FreeRTOS running on ARM cortex-m3," in *Industrial Electronics (ISIE), 2014 IEEE 23rd International Symposium on*, June 2014, pp. 1888–1893.

[9] T. Gomes, P. Garcia, F. Salgado, J. Monteiro, M. Ekpanyapong, and A. Tavares, "Task-aware interrupt controller: Priority space unification in real-time systems," *Embedded Systems Letters, IEEE*, vol. 7, no. 1, pp. 27–30, March 2015.

[10] J. Mistry, M. Naylor, and J. Woodcock, "Adapting freertos for multicores: an experience report," *Software: Practice and Experience*, vol. 44, no. 9, pp. 1129–1154, 2014.

[11] R. Müller, D. Danner, W. Preikschat, and D. Lohmann, "Multi sloth: An efficient multi-core rtos using hardware-based scheduling," in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, July 2014, pp. 189–198.

[12] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, May 1990, pp. 116–123.

[13] S. Pinto, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and A. Tavares, "Towards a lightweight embedded virtualization architecture exploiting arm trustzone," in *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, Sept 2014, pp. 1–4.

Appendix B

GIC Interrupts

The Generic Interrupt Controller provides 96 interrupt lines, with 3 types of interrupts. Table B.1 presents all the Software Generated Interrupts, which are all available to be used, SGI 0 is used for the crosscore communication mechanisms.

Table B.1: Generic Interrupt Controller Interrupt Sources - SGIs

Interrupt ID	Type	Description
0	SGI	Unassigned
1	SGI	Unassigned
2	SGI	Unassigned
3	SGI	Unassigned
4	SGI	Unassigned
5	SGI	Unassigned
6	SGI	Unassigned
7	SGI	Unassigned
8	SGI	Unassigned
9	SGI	Unassigned
10	SGI	Unassigned
11	SGI	Unassigned
12	SGI	Unassigned
13	SGI	Unassigned
14	SGI	Unassigned
15	SGI	Unassigned

Table B.2 presents all the Private Peripheral Interrupts and Shared Peripheral Interrupts. Since all interrupts are either reserved or assigned to a specific hardware peripheral up to SPI number 45, the interrupts utilized for task-associated interrupts are the ones from 46 up to 96, with exception of SPI number 73, which is reserved for the virtual file system (VFS) implemented in the VFS2 component.

Table B.2: Generic Interrupt Controller Interrupt Sources - PPIs and SPIs

Interrupt ID	**Type**	**Description**
16	PPI	Reserved
17	PPI	Reserved
18	PPI	Reserved
19	PPI	Reserved
20	PPI	Reserved
21	PPI	Reserved
22	PPI	Reserved
23	PPI	Reserved
24	PPI	Reserved
25	PPI	Reserved
26	PPI	Reserved
27	PPI	Global Timer
28	PPI	Legacy nFIQ Pin
29	PPI	Private Timer
30	PPI	Watchdog Timer
31	PPI	Legacy nIRQ pin
32	SPI	Watchdog (SP805)
33	SPI	Timer-0
34	SPI	Timer-1
35	SPI	Real-time Clock (PL031)
36	SPI	UART 0
37	SPI	UART 1
38	SPI	UART 2
39	SPI	UART 3 (PL011)
40	SPI	MCI (PL180)
41	SPI	MCI (PL180)
42	SPI	AACI (PL041)
43	SPI	KMI (PL050)
44	SPI	KMI (PL050)
45	SPI	LCD Controller (PL111)
46	SPI	Unassigned
47	SPI	Unassigned
...	...	...
73	SPI	VFS2
74	SPI	Unassigned
75	SPI	Unassigned
...	...	...
96	SPI	Unassigned

Bibliography

[1] D. Lohmann, "Aspect Awareness in the Development of Configurable System Software," Ph.D. dissertation, Universität Erlangen-Nürnberg, 2008.

[2] K. R. Fowler and C. L. Silver, *Developing and managing embedded systems and products*. 225 Wyman Street, USA: Elsevier Inc., 2015.

[3] A. Verma, "Building Secure Systems with ARM TrustZone Technology," *Texas Instrument*, 2013. [Online]. Available: http://www.ti.com/general/docs/lit/getliterature.tsp?baseLiteratureNumber=SPRY228{&}fileType=pdf

[4] Lynx Software Technologies, "LynxOS-178 RTOS : Real-Time Operating Systems." [Online]. Available: http://www.lynx.com/products/real-time-operating-systems/

[5] Richard Barry, *Using the FreeRTOS Real-Time Kernel*, 2010.

[6] W. Hofer, "SLOTH: The Virtue and Vice of Latency Hiding," Ph.D. dissertation.

[7] L. E. Leyva-Del-Foyo, P. Mejia-Alvarez, and D. De Niz, "Predictable interrupt management for real time kernels over conventional PC hardware," in *IEEE Real-Time and Embedded Technology and Applications (RTAS '06)*, 2006, pp. 14–23.

[8] F. Scheler, W. Hofer, B. Oechslein, R. Pfister, W. Schröder-Preikschat, and D. Lohmann, "Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system," *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems - CASES '09*, p. 167, 2009.

[9] Wanja Hofer, D. Lohmann, F. Scheler, W. Schröder, and Preikschat, "Sloth: Threads as Interrupts as Threads," in *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS), IEEE Computer Society*, no. D, 2009, pp. 204 – 213.

[10] ARM, "Cortex A9 Processor." [Online]. Available: http://www.arm.com/products/processors/cortex-a/cortex-a9.php

[11] R. Müller, D. Danner, W. Schröder-Preikschat, and D. Lohmann, "MULTI-SLOTH: An Efficient Multi-Core RTOS using Hardware-Based Scheduling," in *ECRTS*, 2014, pp. 189–198.

[12] A. S. P. Galvin, G. Gagne, *Operating System Concepts*, 8th ed. Wiley, 2005.

[13] S. Pinto, "Sistema Operativo Orientado a Objetos : porting , expansão e configuração," Ph.D. dissertation, Universidade do Minho, 2012.

[14] R. Oshana, *Software Engineering of Embedded and Real-Time Systems.* Elsevier Inc., 2013.

[15] A. Sloss, D. Symes, and C. Wright, *ARM system developer's guide: designing and optimizing system software*, 2004.

[16] T. Noergaard, *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*, 2005.

[17] D. B. Stewart, "Twenty-Five Most Common Mistakes with Real-Time Software Development," *1999 Embedded Systems Conference*, no. September, 1999.

[18] S. Kleiman and J. Eykholt, "Interrupts as threads," *ACM SIGOPS Operating Systems Review*, vol. 29, pp. 21–26, 1995.

[19] D. Lohmann and J. Streicher, "Interrupt Synchronization in the CiAO Operating System," in *Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patterns for Infras- tructure Software (AOSD-ACP4IS)*, 2007.

[20] S. Chandra, F. Regazzoni, and M. Lajolo, "Hardware / Software Partitioning of Operating Systems : a Behavioral Synthesis Approach," *Proceeding GLSVLSI 'Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pp. 324–329, 2006.

[21] Di-Shi Sun, Douglas M. Blough and V. J. M. III, "Atalanta: A new multiprocessor RTOS kernel for system-on-a-chip applications," *Technical report, Georgia Institute of Technology*, pp. 1–16, 2002.

[22] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai, "Hardware implementation of a real-time operating system," *Proceedings of the 12th TRON Project International Symposium*, 1995.

[23] T. Gomes, P. Garcia, F. Salgado, J. Monteiro, M. Ekpanyapong, and A. Tavares, "Task-Aware Interrupt Controller: Priority Space Unification in Real-Time Systems," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 27–30, 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7024904

[24] S. Pinto, J. Pereira, D. Oliveira, F. Alves, E. Qaralleh, M. Ekpanyapong, J. Cabral, and A. Tavares, "Porting SLOTH system to FreeRTOS running on ARM Cortex-M3," *2014 IEEE 23rd International Symposium on Industrial Electronics (ISIE)*, pp. 1888–1893, 2014.

[25] W. Hofer, D. Lohmann, and W. Schröder-Preikschat, "SLEEPY SLOTH: Threads as interrupts as threads," in *Proceedings - Real-Time Systems Symposium*, 2011, pp. 67–77.

[26] D. Danner, R. Müller, W. Schröder-preikschat, W. Hofer, D. Lohmann, F. Alexander, U. Fau, and E. Nürnberg, "SAFER SLOTH : Efficient , Hardware-Tailored Memory Protection," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, 2014, pp. 37–48.

[27] Nvidia, "The Benefits of Multiple CPU Cores in Mobile Devices," *NVIDIA Corporation*, pp. 1–23, 2010.

[28] B. Moyer, *Real World Multicore Embedded Systems*. Elsevier Inc., 2013.

[29] A. R. M. Limited, "ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition," 2012.

[30] ARM, "PrimeCell Generic Interrupt Controller," 2009.

[31] A. R. M. Cortex, "Cortex-A9 MPCore Technical Reference Manual," 2009.

[32] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," *Proceedings.,10th International Conference on Distributed Computing Systems*, 1990.

[33] ARM, "Fixed Virtual Platforms - FVP Reference Guide," pp. 1–49, 2014.

[34] S. Pinto, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and a. Tavares, "Towards a Lightweight Embedded Virtualization Architecture Exploiting ARM TrustZone," *Emerging Technology and Factory Automation*, p. 4, 2014.

[35] Express Logic, "Measuring RTOS Performance." [Online]. Available: http://rtos.com/PDFs/MeasuringRTOSPerformance.pdf

[36] Xilinx, "ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC," *User Guide*, vol. 850, 2012.